The Python Language Reference

リリース *3.13.0rc2*

Guido van Rossum and the Python development team

9 月 19, 2024

目次

このリファレンスマニュアルでは、Python 言語の文法と、” コアとなるセマンティクス” について記述します。このマニュアルはそっけない書き方かもしれませんが、正確さと完全さを優先しています。必須でない組み込みオブジェクト型や組み込み関数、組み込みモジュールに関するセマンティクスは、library-index で述べられています。形式ばらない Python 言語入門には、tutorial-index を参照してください。C 言語あるいは C++ プログラマ向けには、このマニュアルとは別に二つのマニュアルがあります。extending-index では、Python 拡張モジュールを書くための高レベルな様式について述べています。また、c-api-index では、C/C++ プログラマが利用できるインターフェースについて詳細に記述しています。

第

ONE

はじめに

このリファレンスマニュアルは、Python プログラミング言語自体に関する記述です。チュートリアルとして書かれたものではありません。

私は本マニュアルをできるだけ正確に書こうとする一方で、文法や字句解析以外の全てについて、形式化された仕様記述ではなく英語を使うことにしました。そうすることで、このドキュメントが平均的な読者にとってより読みやすくなっているはずですが、ややあいまいな部分も残っていることでしょう。従って、もし読者のあなたが火星から来ている人で、このドキュメントだけから Python を再度実装しようとしているのなら、色々と推測しなければならないことがあり、実際にはおそらく全く別の言語を実装する羽目になるでしょう。逆に、あなたが Python を利用しており、Python 言語のある特定の領域において、厳密な規則が何か疑問に思った場合、その答えはこのドキュメントで確実に見つけられることでしょう。もしより形式化された言語定義をお望みなら、あなたの時間を提供していただいてかまいません --- もしくは、クローン生成装置でも発明してください :-)。

実装に関する詳細を言語リファレンスのドキュメントに載せすぎるのは危険なことです --- 実装は変更されるかもしれず、同じ言語でも異なる実装は異なった動作をするかもしれないからです。一方、CPython が広く使われている一つの Python 実装 (別の実装も支持され続けていますが) なので、特定のクセについては、特に実装によって何らかの制限が加えられている場合には、触れておく価値があります。従って、このテキスト全体にわたって短い ” 実装に関する注釈 (impementation notes)” がちりばめられています。

Python 実装はいずれも、数々の組み込みモジュールと標準モジュールが付属します。それらについては、library-index でドキュメント化されています。いくつかの組み込みモジュールについては、言語定義と重要なかかわりをもっているときについて触れています。

1.1 別の Python の実装

Python の実装としては、群を抜いて有名な実装がひとつ存在しています。それ以外の実装に関しても、特定のユーザ間で興味が持たれています。

よく知られている実装には以下のものがあります:

CPython
: これは最も保守されている初代の Python 実装で、C 言語で書かれています。ほとんどの場合、言語の新機能

がいち早く実装されます。

Jython
: Java で実装された Python です。この実装は Java アプリケーションのためのスクリプト言語として、もしくは Java クラスライブラリを使ったアプリケーションを作成するために使用することができます。また、Java ライブラリのテストを作成するためにもしばしば使用されています。さらなる情報については the Jython website を参照してください。

Python for .NET
: この実装は内部では CPython を使用していますが、.NET アプリケーションによって管理されているので、.NET ライブラリを参照することが可能です。この実装は Brian Lloyd によって作成されました。さらなる情報については、Python for .NET home page を参照してください。

IronPython
: .NET で Python を使用するためのもう一つの実装です。Python.NET とは異なり、完全に IL を生成することができる Python の実装あり、直接 Python コードを.NET アセンブリにコンパイルします。これは Jython の初代の開発者である Jim Hugunin によって作られました。さらなる情報については the IronPython website を参照してください。

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

これらの各実装はこのマニュアルで文書化された言語とは多少異なっている、もしくは、標準の Python ドキュメントと何処が異なっているかを定めた情報が公開されているでしょう。あなたが使用している実装上で、代替手段を使う必要があるかどうかを判断するためには、各実装の仕様書を参照してください。

1.2 本マニュアルにおける表記法

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

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

最初の行は、`name` が `lc_letter` の後ろにゼロ個またはそれ以上の `lc_letter` とアンダースコアが続いたものであることを示しています。そして、`lc_letter` は `'a'` から `'z'` までの何らかの文字一字であることを示します。(この規則は、このドキュメントに記述されている字句規則と構文規則において定義されている名前 (name)

で一貫して使われています)。

各規則は name (規則によって定義されているものの名前) と ::= から始まります。垂直線 (|) は、複数の選択項目を分かち書きするときに使います; この記号は、この記法において最も結合優先度の低い演算子です。アスタリスク (*) は、直前にくる要素のゼロ個以上の繰り返しを表します; 同様に、プラス (+) は一個以上の繰り返しで、角括弧 ([ ]) に囲われた字句は、字句がゼロ個か一個出現する (別の言い方をすれば、囲いの中の字句はオプションである) ことを示します。* および + 演算子の結合範囲は可能な限り狭くなっています; 字句のグループ化には丸括弧を使います。リテラル文字列はクオートで囲われます。空白はトークンを分割しているときのみ意味を持ちます。規則は通常、一行中に収められています; 多数の選択肢のある規則は、最初の行につづいて、垂直線の後ろに各々別の行として記述されます。

(上の例のような) 字句定義では、他に二つの慣習が使われています: 三つのドットで区切られている二つのリテラル文字は、二つの文字の ASCII 文字コードにおける (包含的な) 範囲から文字を一字選ぶことを示します。各カッコ中の字句 (<...>) は、定義済みのシンボルを記述する非形式的なやりかたです; 例えば、' 制御文字' を書き表す必要があるときなどに使われることがあります。

字句と構文規則の定義の間で使われている表記はほとんど同じですが、その意味には大きな違いがあります: 字句定義は入力ソース中の個々の文字を取り扱いますが、構文定義は字句解析で生成された一連のトークンを取り扱います。次節 (" 字句解析") における BNF はすべて字句定義のためのものです; それ以降の章では、構文定義のために使っています。

第

TWO

字句解析

Python で書かれたプログラムは **パーザ** *(parser)* に読み込まれます。パーザへの入力は、**字句解析器** *(lexical analyzer)* によって生成された一連の **トークン** *(token)* からなります。この章では、字句解析器がファイルをトークン列に分解する方法について解説します。

Python はプログラムテキストを Unicode コードポイントとして読み込みます。ソースファイルのエンコーディングはエンコーディング宣言で与えられ、デフォルトは UTF-8 です。詳細は **PEP 3120** を参照してください。ソースファイルがデコードできなければ、`SyntaxError` が送出されます。

2.1 行構造

Python プログラムは多数の **論理行** *(logical lines)* に分割されます。

2.1.1 論理行 (logical line)

論理行の終端は、トークン NEWLINE で表されます。構文上許されている場合 (複合文: compound statement 中の実行文: statement) を除いて、実行文は論理行間にまたがることはできません。論理行は一行またはそれ以上の **物理行** *(physical line)* からなり、物理行の末尾には明示的または非明示的な **行連結** *(line joining)* 規則が続きます。

2.1.2 物理行 (physical line)

物理行とは、行終端コードで区切られた文字列のことです。ソースファイルやソース文字列では、各プラットフォームごとの標準の行終端コードを使用することができます。Unix 形式では ASCII LF (行送り: linefeed) 文字、Windows 形式では ASCII 配列の CR LF (復帰: return に続いて行送り) 、Macintosh 形式では ASCII CR (復帰) 文字です。これら全ての形式のコードは、違うプラットフォームでも等しく使用することができます。入力の末尾も、最後の物理行の暗黙的な終端としての役割を果たします。

Python に埋め込む場合には、標準の C 言語の改行文字の変換規則 (ASCII LF を表現した文字コード `\n` が行終端となります) に従って、Python API にソースコードを渡す必要があります。

2.1.3 コメント (Comments)

コメントは文字列リテラル内に入っていないハッシュ文字 (`#`) から始まり、同じ物理行の末端で終わります。非明示的な行継続規則が適用されていない限り、コメントは論理行を終端させます。コメントは構文上無視されます。

2.1.4 エンコード宣言 (encoding declaration)

Python スクリプト中の一行目か二行目にあるコメントが正規表現 `coding[=:]\s*([-\w.]+)` にマッチする場合、コメントはエンコード宣言として処理されます; この表現の最初のグループがソースコードファイルのエンコードを指定します。エンコード宣言は自身の行になければなりません。二行目にある場合、一行目もコメントのみの行でなければなりません。エンコード宣言式として推奨する形式は

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

これは GNU Emacs で認識できます。または

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

これは、Bram Moolenar による VIM が認識できる形式です。

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

エンコーディングが宣言される場合、そのエンコーディング名は Python によって認識できなければなりません (standard-encodings を参照してください)。宣言されたエンコーディングは、例えば文字列リテラル、コメント、識別子などの、全ての字句解析に使われます。

2.1.5 明示的な行継続

二つまたはそれ以上の物理行を論理行としてつなげるためには、バックスラッシュ文字 (\) を使って以下のようにします: 物理行が文字列リテラルやコメント中の文字でないバックスラッシュで終わっている場合、後続する行とつなげて一つの論理行を構成し、バックスラッシュおよびバックスラッシュの後ろにある行末文字を削除します。例えば:

```
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
```

バックスラッシュで終わる行にはコメントを入れることはできません。また、バックスラッシュを使ってコメントを継続することはできません。バックスラッシュが文字列リテラル中にある場合を除き、バックスラッシュの後ろにトークンを継続することはできません (すなわち、物理行内の文字列リテラル以外のトークンをバックスラッシュを使って分断することはできません)。上記以外の場所では、文字列リテラル外にあるバックスラッシュはどこにあっても不正となります。

2.1.6 非明示的な行継続

丸括弧 (parentheses)、角括弧 (square bracket) 、および波括弧 (curly brace) 内の式は、バックスラッシュを使わずに一行以上の物理行に分割することができます。例えば:

```
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
```

非明示的に継続された行にはコメントを含めることができます。継続行のインデントは重要ではありません。空の継続行を書くことができます。非明示的な継続行中には、NEWLINE トークンは存在しません。非明示的な行の継続は、三重クオートされた文字列 (下記参照) でも発生します; この場合には、コメントを含めることができません。

2.1.7 空行

スペース、タブ、フォームフィード、およびコメントのみを含む論理行は無視されます (すなわち、NEWLINE トークンは生成されません)。文を対話的に入力している際には、空行の扱いは行読み込み-評価-出力 (read-eval-print) ループの実装によって異なることがあります。標準的な対話的インタプリタの実装では、完全な空行でできた論理行 (すなわち、空白文字もコメントも全く含まない空行) は、複数行からなる文の終端を示します。

2.1.8 インデント

論理行の行頭にある、先頭の空白 (スペースおよびタブ) の連なりは、その行のインデントレベルを計算するために使われます。インデントレベルは、実行文のグループ化方法を決定するために用いられます。

タブは (左から右の方向に) 1 つにつき 8 つのスペースで置き換えられ、置き換え後の文字数は 8 の倍数になります (Unix で使われている規則と同じになるよう意図されています)。そして、最初の非空白文字までのスペースの総数が、その行のインデントを決定します。インデントは、バックスラッシュで複数の物理行に分割できません; 最初のバックスラッシュまでの空白がインデントを決定します。

ソースファイルがタブとスペースを混在させ、その意味づけがタブのスペース換算数に依存するようなら、インデントは不合理なものとして却下されます。その場合は `TabError` が送出されます。

プラットフォーム間の互換性に関する注意: 非 UNIX プラットフォームにおけるテキストエディタの性質上、一つのソースファイル内でタブとインデントを混在させて使うのは賢明ではありません。また、プラットフォームによっては、最大インデントレベルを明示的に制限しているかもしれません。

フォームフィード文字が行の先頭にあっても構いません; フォームフィード文字は上のインデントレベル計算時には無視されます。フォームフィード文字が先頭の空白中の他の場所にある場合、その影響は未定義です (例えば、スペースの数を 0 にリセットするかもしれません)。

連続する行における各々のインデントレベルは、INDENT および DEDENT トークンを生成するために使われます。トークンの生成はスタックを用いて以下のように行われます。

ファイル中の最初の行を読み出す前に、スタックにゼロが一つ積まれ (push され) ます; このゼロは決して除去 (pop) されることはありません。スタックの先頭に積まれてゆく数字は、常にスタックの末尾から先頭にかけて厳密に増加するようになっています。各論理行の開始位置において、その行のインデントレベル値がスタックの先頭の値と比較されます。値が等しければ何もしません。インデントレベル値がスタック上の値よりも大きければ、インデントレベル値はスタックに積まれ、INDENT トークンが一つ生成されます。インデントレベル値がスタック上の値よりも小さい場合、その値はスタック内のいずれかの値と **等しくなければなりません** ; スタック上のインデントレベル値よりも大きい値はすべて除去され、値が一つ除去されるごとに DEDENT トークンが一つ生成されます。ファイルの末尾では、スタックに残っているゼロより大きい値は全て除去され、値が一つ除去されるごとに DEDENT トークンが一つ生成されます。

以下の例に正しく (しかし当惑させるように) インデントされた Python コードの一部を示します:

```
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
```

以下の例は、様々なインデントエラーになります:

```
 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
```

(実際は、最初の 3 つのエラーはパーザによって検出されます; 最後のエラーのみが字句解析器で見つかります ---

return r のインデントは、スタックから逐次除去されていくどのインデントレベル値とも一致しません)

2.1.9 トークン間の空白

論理行の先頭や文字列の内部にある場合を除き、空白文字であるスペース、タブ、およびフォームフィードは、トークンを分割するために自由に利用することができます。二つのトークンを並べて書くと別のトークンとしてみなされてしまうような場合には、トークンの間に空白が必要となります (例えば、ab は一つのトークンですが、a b は二つのトークンとなります)。

2.2 その他のトークン

NEWLINE、INDENT、および DEDENT の他、以下のトークンのカテゴリ: **識別子** *(identifier)*, **キーワード** *(keyword)*, **リテラル**, **演算子** *(operator)*, **デリミタ** *(delimiter)* が存在します。空白文字 (上で述べた行終端文字以外) はトークンではありませんが、トークンを区切る働きがあります。トークンの解析にあいまいさが生じた場合、トークンは左から右に読んで不正でないトークンを構築できる最長の文字列を含むように構築されます。

2.3 識別子 (identifier) およびキーワード (keyword)

識別子 (または **名前** *(name)*) は、以下の字句定義で記述されます。

Python における識別子の構文は、Unicode 標準仕様添付書類 UAX-31 に基づき、詳細と変更点は以下で定義します。詳しくは **PEP 3131** を参照してください。

ASCII 範囲 (U+0001..U+007F) 内では、識別子として有効な文字は Python 2.x におけるものと同じです。大文字と小文字の A から Z、アンダースコア _、先頭の文字を除く数字 0 から 9 です。

Python 3.0 は、さらに ASCII 範囲外から文字を導入します (**PEP 3131** を参照してください。)。これらの文字については、分類は unicodedata モジュールに含まれる Unicode Character Database の版を使います。

識別子の長さには制限がありません。大小文字は区別されます。

```
identifier    ::=  xid_start xid_continue*
id_start      ::=  <all characters in general categories Lu, Ll, Lt, Lm, Lo, Nl, the underscore, and
id_continue   ::=  <all characters in id_start, plus characters in the categories Mn, Mc, Nd, Pc and
xid_start     ::=  <all characters in id_start whose NFKC normalization is in "id_start xid_continue*
xid_continue  ::=  <all characters in id_continue whose NFKC normalization is in "id_continue*">
```

上で言及した Unicode カテゴリコードは以下を表します:

- *Lu* - 大文字 (uppercase letters)

- *Ll* - 小文字 (lowercase letters)
- *Lt* - 先頭が大文字 (titlecase letters)
- *Lm* - 修飾文字 (modifier letters)
- *Lo* - その他の文字 (other letters)
- *Nl* - 数値を表す文字 (letter numbers)
- *Mn* - 字幅のない記号 (nonspacing marks)
- *Mc* - 字幅のある結合記号 (spacing combining marks)
- *Nd* - 10 進数字 (decimal numbers)
- *Pc* - 連結用句読記号 (connector punctuations)
- *Other_ID_Start* - explicit list of characters in PropList.txt to support backwards compatibility
- *Other_ID_Continue* - 同様

すべての識別子は、解析中は正規化形式 NFKC に変換されます。識別子間の比較は NFKC に基づきます。

A non-normative HTML file listing all valid identifier characters for Unicode 15.1.0 can be found at https://www.unicode.org/Public/15.1.0/ucd/DerivedCoreProperties.txt

2.3.1 キーワード (keyword)

以下の識別子は、予約語、または Python 言語における **キーワード** *(keyword)* として使われ、通常の識別子として使うことはできません。キーワードは厳密に下記の通りに綴らなければなりません:

```
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 ソフトキーワード

Added in version 3.10.

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

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

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

バージョン 3.12 で変更: `type` is now a soft keyword.

2.3.3 予約済みの識別子種 (reserved classes of identifiers)

ある種の (キーワードを除く) 識別子には、特殊な意味があります。これらの識別子種は、先頭や末尾にあるアンダースコア文字のパターンで区別されます:

`_*`
: Not imported by `from module import *`.

`_`
: In a `case` pattern within a *`match`* statement, `_` is a *soft keyword* that denotes a *wildcard*.

  Separately, the interactive interpreter makes the result of the last evaluation available in the variable `_`. (It is stored in the `builtins` module, alongside built-in functions like `print`.)

  Elsewhere, `_` is a regular identifier. It is often used to name "special" items, but it is not special to Python itself.

  注釈

  名前 `_` は、しばしば国際化 (internationalization) と共に用いられます; この慣習についての詳しい情報は、`gettext` を参照してください。

  It is also commonly used for unused variables.

`__*__`
: システムで定義された (system-defined) 名前です。非公式には"dunder" な名前と呼ばれます (訳注: double underscores の略)。これらの名前はインタープリタと (標準ライブラリを含む) 実装上で定義されています。現行のシステムでの名前は **特殊メソッド名** などで話題に挙げられています。Python の将来のバージョンではより多くの名前が定義されることになります。このドキュメントで明記されている用法に従わない、**あ**

らゆる `__*__` の名前は、いかなるコンテキストにおける利用でも、警告無く損害を引き起こすことがあります。

`__*` クラスプライベート (class-private) な名前です。このカテゴリに属する名前は、クラス定義のコンテキスト上で用いられた場合、基底クラスと派生クラスの ” プライベートな” 属性間で名前衝突が起こるのを防ぐために書き直されます。**識別子** *(identifier、* **または名前** *(name))* を参照してください。

2.4 リテラル

リテラル (literal) とは、いくつかの組み込み型の定数を表記したものです。

2.4.1 文字列およびバイト列リテラル

文字列リテラルは以下の字句定義で記述されます:

```
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>
```

上記の生成規則で示されていない文法的な制限が一つあります。リテラルの *stringprefix* や *bytesprefix* と残りの部分の間に空白を入れてはならないことです。ソースコード文字セット (source character set) はエンコーディング宣言で定義されます。エンコーディング宣言がなければ UTF-8 です。節 **エンコード宣言** *(encoding declaration)* を参照してください。

In plain English: Both types of literals can be enclosed in matching single quotes (') or double quotes ("). They can also be enclosed in matching groups of three single or double quotes (these are generally referred to as *triple-quoted strings*). The backslash (\) character is used to give special meaning to otherwise ordinary characters like `n`, which means 'newline' when escaped (`\n`). It can also be used to escape characters that otherwise have a special meaning, such as newline, backslash itself, or the quote character. See *escape sequences* below for examples.

バイト列リテラルには、常に `'b'` や `'B'` が接頭します。これらによって、`str` 型ではなく `bytes` 型のインスタンスが作成されます。バイト列リテラルは ASCII 文字のみ含むことができます。128 以上の数値を持つバイトはエスケープして表されなければなりません。

Both string and bytes literals may optionally be prefixed with a letter `'r'` or `'R'`; such constructs are called *raw string literals* and *raw bytes literals* respectively and treat backslashes as literal characters. As a result, in raw string literals, `'\U'` and `'\u'` escapes are not treated specially.

Added in version 3.3: raw バイト列リテラルの `'rb'` プレフィックスが `'br'` の同義語として追加されました。

Python 2.x と 3.x 両対応のコードベースのメンテナンスを単純化するために、レガシー unicode リテラル (`u'value'`) のサポートが再び導入されました。詳細は **PEP 414** を参照してください。

`'f'` または `'F'` の接頭辞が付いた文字列リテラルはフォーマット済み文字列リテラル（ *formatted string literal* ）です。詳細については **フォーマット済み文字列リテラル** を参照してください。接頭辞の `'f'` は `'r'` と組み合わせられますが、`'b'` や `'u'` と組み合わせることはできません。つまりフォーマット済みの raw 文字列リテラルは可ですが、フォーマット済みのバイト列リテラルは不可です。

三重クオートリテラル中には、三連のエスケープされないクオート文字でリテラルを終端してしまわないかぎり、エスケープされていない改行やクオートを書くことができます (さらに、それらはそのまま文字列中に残ります)。(ここでいう ”クオート” とは、文字列の囲みを開始するときに使った文字を示し、`'` か `"` のいずれかです。)

Escape sequences

'r' または 'R' 接頭文字がつかないかぎり、文字列またはバイト列リテラル中のエスケープシーケンスは標準 C で使われているのと同様の法則にしたがって解釈されます。以下に Python で認識されるエスケープシーケンスを示します:

エスケープシーケンス	意味	注釈
\<newline>	バックスラッシュと改行文字が無視されます	(1)
\\	バックスラッシュ (\)	
\'	一重引用符 (')	
\"	二重引用符 (")	
\a	ASCII 端末ベル (BEL)	
\b	ASCII バックスペース (BS)	
\f	ASCII フォームフィード (FF)	
\n	ASCII 行送り (LF)	
\r	ASCII 復帰 (CR)	
\t	ASCII 水平タブ (TAB)	
\v	ASCII 垂直タブ (VT)	
ooo	8 進数値 *ooo* を持つ文字	(2,4)
\x*hh*	16 進数値 *hh* を持つ文字	(3,4)

文字列でのみ認識されるエスケープシーケンスは以下のとおりです:

エスケープシーケンス	意味	注釈
\N{*name*}	Unicode データベース中で *name* という名前の文字	(5)
\u*xxxx*	16-bit の十六進値 *xxxx* を持つ文字	(6)
\U*xxxxxxxx*	32-bit の十六進値 *xxxxxxxx* を持つ文字	(7)

注釈:

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

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

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

(2) 標準 C と同じく、最大で 3 桁の 8 進数まで受理します。

バージョン 3.11 で変更: Octal escapes with value larger than `0o377` produce a `DeprecationWarning`.

バージョン 3.12 で変更: Octal escapes with value larger than `0o377` produce a `SyntaxWarning`. In a future Python version they will be eventually a `SyntaxError`.

(3) 標準 C とは違い、ちょうど 2 桁の 16 進数しか受理されません。

(4) バイト列リテラル中では、十六進および八進エスケープは与えられた値のバイトを表します。文字列リテラル中では、エスケープ文字は与えられた値を持つ Unicode 文字を表します。

(5) バージョン 3.3 で変更: name aliases[*1] に対するサポートが追加されました。

(6) ちょうど 4 桁の 16 進数しか受理されません。

(7) あらゆるユニコード文字はこのようにしてエンコードすることができます。正確に 8 文字の 16 進数字が必要です。

標準の C とは違い、認識されなかったエスケープシーケンスはすべて、そのまま文字列中に残ります。すなわち、**バックスラッシュも結果中に残ります**。(この挙動はデバッグの際に便利です: エスケープシーケンスが誤入力されたら、その出力結果が失敗しているのが分かりやすくなります。) 文字列中でのみ認識されるエスケープシーケンスは、バイト列リテラルには、認識されないエスケープシーケンスとして分類されるので注意してください。

バージョン 3.6 で変更: Unrecognized escape sequences produce a `DeprecationWarning`.

バージョン 3.12 で変更: Unrecognized escape sequences produce a `SyntaxWarning`. In a future Python version they will be eventually a `SyntaxError`.

raw リテラルでも、引用符はバックスラッシュでエスケープできますが、バックスラッシュ自体も文字列に残ります; 例えば、`r"\""` は有効な文字列リテラルで、バックスラッシュと二重引用符からなる文字列を表します; `r"\"` は無効な文字列リテラルです (raw リテラルを奇数個連なったバックスラッシュで終わらせることはできません)。具体的には、(バックスラッシュが直後のクオート文字をエスケープしてしまうので) *raw* **文字列を単一のバックスラッシュで終わらせることはできません** さらに、バックスラッシュの直後に改行がきても、行継続を意味する**のではなく**、リテラルの一部であるそれら二つの文字として解釈されます。

2.4.2 文字列リテラルの結合 (concatenation)

文字列やバイト列リテラルは、互いに異なる引用符を使っていても (空白文字で区切っても) 複数隣接させることができます。これは各々の文字列を結合するのと同じ意味を持ちます。したがって、`"hello" 'world'` は `"helloworld"` と同じです。この機能を使うと、バックスラッシュを減らしたり、長い文字列を手軽に分離して複数行にまたがらせたり、あるいは部分文字列ごとにコメントを追加することさえできます。例えば:

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

[*1] https://www.unicode.org/Public/15.1.0/ucd/NameAliases.txt

この機能は文法レベルで定義されていますが、スクリプトをコンパイルする際の処理として実現されることに注意してください。実行時に文字列表現を結合したければ、'+' 演算子を使わなければなりません。また、リテラルの結合においては、結合する各要素に異なる引用符形式を使ったり (raw 文字列と三重引用符を混ぜることさえできます) 、フォーマット済み文字列リテラルと通常の文字列リテラルを結合したりすることもできますので注意してください。

2.4.3 f-strings

Added in version 3.6.

フォーマット済み文字列リテラル（ *formatted string literal* ）または *f-string* は、接頭辞 `'f'` または `'F'` の付いた文字列リテラルです。これらの文字列には、波括弧 `{}` で区切られた式である置換フィールドを含めることができます。他の文字列リテラルの場合は内容が常に一定で変わることが無いのに対して、フォーマット済み文字列リテラルは実行時に式として評価されます。

エスケープシーケンスは通常の文字列リテラルと同様にデコードされます (ただしリテラルが raw 文字列でもある場合は除きます) 。エスケープシーケンスをデコードした後は、文字列の内容は次の文法で解釈されます:

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

文字列のうち、波括弧で囲まれた部分以外は文字通り解釈されます。ただし、二重波括弧 `'{{'` および `'}}'` は単一の波括弧に置き換えられます。単一の開き波括弧 `'{'` は置換フィールドの始まりを意味し、その中身は Python の式で始まります。(デバッグ時に便利な機能として) 式のテキストと、評価後の値との両者を表示したい場合には、式の後に等号 `'='` を加えてください。その後ろには、感嘆符 `'!'` によって導入される変換フィールドを続けることができます。さらに、`':'` に続いて書式指定子を追加できます。置換フィールドは単一の閉じ波括弧 `'}'` で終わります。

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

must be closed in a different line.

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

バージョン 3.7 で変更: Python 3.7 より前のバージョンでは、*await* 式および *async for* 句を含む内包表記は、実装に伴う問題の都合により許されていませんでした。

バージョン 3.12 で変更: Prior to Python 3.12, comments were not allowed inside f-string replacement fields.

等号 `'='` が指定されたとき、出力文字列は、式のテキスト表現、等号 `'='`、および評価された式を含みます。開き波括弧 `'{'` の直後、式の中、および `'='` の後に含まれる空白文字はすべて保存されます。書式指定子が存在しない限り、`'='` を指定した場合は、式に対して `repr()` を適用した結果が出力になります。一方、書式指定子が存在する場合は、変換フィールドで `'!r'` が指定されていない限り、デフォルトで `str()` が適用されます。

Added in version 3.8: 等号 `'='`。

もし変換フィールドが指定されていた場合、式の評価結果はフォーマットの前に変換されます。変換 `'!s'` は `str()` を、`'!r'` は `repr()` を、そして `'!a'` は `ascii()` を呼び出します。

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

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

フォーマット済み文字列リテラルは他の文字列リテラルと結合できますが、置換フィールドを複数のリテラルに分割して書くことはできません。

フォーマット済み文字列リテラルの例をいくつか挙げます:

```
>>> 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'
```

(次のページに続く)

(前のページからの続き)

```
>>> f"{today=:%B %d, %Y}" # using date format specifier and debugging
'today=January 27, 2017'
>>> number = 1024
>>> f"{number:#0x}"  # using integer format specifier
'0x400'
>>> foo = "bar"
>>> f"{ foo = }" # preserves whitespace
" foo = 'bar'"
>>> line = "The mill's closed"
>>> f"{line = }"
'line = "The mill\'s closed"'
>>> f"{line = :20}"
"line = The mill's closed   "
>>> f"{line = !r:20}"
'line = "The mill\'s closed" '
```

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

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

バージョン 3.12 で変更: Prior to Python 3.12, reuse of the same quoting type of the outer f-string inside a replacement field was not possible.

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

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

バージョン 3.12 で変更: Prior to Python 3.12, backslashes were not permitted inside an f-string replacement field.

フォーマット済み文字列リテラルは、たとえ式を含んでいなかったとしても、docstring としては使えません。

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

フォーマット済み文字列リテラルを Python に追加した提案 **PEP 498** も参照してください。また関連する文字列フォーマットの仕組みを使っている `str.format()` も参照してください。

2.4.4 数値リテラル

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

数値リテラルには符号が含まれていないことに注意してください; `-1` のような句は、実際には単項演算子 (unary operator) '`-`' とリテラル `1` を組み合わせたものです。

2.4.5 整数リテラル

整数リテラルは以下の字句定義で記述されます:

```
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"
```

値がメモリ上に収まるかどうかという問題を除けば、整数リテラルには長さの制限がありません。

アンダースコアはリテラルの値を判断するにあたって無視されます。そのためアンダースコアを使って数字をグループ化することで読みやすくできます。アンダースコアは数字と数字の間に 1 つだけ、あるいは `0x` のような基数指定の直後に 1 つだけ挿入できます。

なお、非 0 の十進数の先頭には 0 を付けられません。これは、Python がバージョン 3.0 以前に使っていた C 形式の八進リテラルとの曖昧さを回避するためです。

整数リテラルの例をいくつか示します:

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

バージョン 3.6 で変更: グループ化を目的としたリテラル中のアンダースコアが許されるようになりました。

2.4.6 Floating-point literals

Floating-point literals are described by the following lexical definitions:

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

Note that the integer and exponent parts are always interpreted using radix 10. For example, `077e010` is legal, and denotes the same number as `77e10`. The allowed range of floating-point literals is implementation-dependent. As in integer literals, underscores are supported for digit grouping.

Some examples of floating-point literals:

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

バージョン 3.6 で変更: グループ化を目的としたリテラル中のアンダースコアが許されるようになりました。

2.4.7 虚数 (imaginary) リテラル

虚数リテラルは以下のような字句定義で記述されます:

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

An imaginary literal yields a complex number with a real part of 0.0. Complex numbers are represented as a pair of floating-point numbers and have the same restrictions on their range. To create a complex number with a nonzero real part, add a floating-point number to it, e.g., `(3+4j)`. Some examples of imaginary literals:

```
3.14j   10.j    10j     .001j   1e100j   3.14e-10j   3.14_15_93j
```

2.5 演算子

以下のトークンは演算子です:

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

2.6 デリミタ (delimiter)

以下のトークンは文法上のデリミタとして働きます:

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

ピリオドは浮動小数点数や虚数リテラル中にも置けます。ピリオド三つの列はスライス表記における省略符号 (ellipsis) リテラルとして特別な意味を持ちます。リスト後半の累算代入演算子 (augmented assignment operator) は、字句的にはデリミタとして振舞いますが、演算も行います。

以下の印字可能 ASCII 文字は、他のトークンの一部として特殊な意味を持っていたり、字句解析器にとって重要な意味を持っています:

```
'       "       #       \
```

以下の印字可能 ASCII 文字は、Python では使われていません。これらの文字が文字列リテラルやコメントの外にある場合、無条件にエラーとなります:

```
$       ?       `
```

脚注

第

THREE

データモデル

3.1 オブジェクト、値、および型

Python における **オブジェクト** *(object)* とは、データを抽象的に表したものです。Python プログラムにおけるデータは全て、オブジェクトまたはオブジェクト間の関係として表されます。(ある意味では、プログラムコードもまたオブジェクトとして表されます。これはフォン・ノイマン: Von Neumann の ” プログラム記憶方式コンピュータ: stored program computer” のモデルに適合します。)

すべてのオブジェクトは、同一性 (identity)、型、値をもっています。**同一性** は生成されたあとは変更されません。これはオブジェクトのアドレスのようなものだと考えられるかもしれません。*is* 演算子は 2 つのオブジェクトの同一性を比較します。`id()` 関数は同一性を表す整数を返します。

CPython 実装の詳細: CPython では、`id(x)` は `x` が格納されているメモリ上のアドレスを返します。

オブジェクトの型はオブジェクトがサポートする操作 (例: `len()` をサポートするか) と、オブジェクトが取りうる値を決定します。`type()` 関数はオブジェクトの型 (型自体もオブジェクトです) を返します。同一性と同じく、オブジェクトの型 (*type*) も変更不可能です。[*1]

オブジェクトによっては **値** を変更することが可能です。値を変更できるオブジェクトのことを *mutable* と呼びます。生成後に値を変更できないオブジェクトのことを *immutable* と呼びます。(mutable なオブジェクトへの参照を格納している immutable なコンテナオブジェクトの値は、その格納しているオブジェクトの値が変化した時に変化しますが、コンテナがどのオブジェクトを格納しているのかが変化しないのであれば immutable だと考えることができます。したがって、immutable かどうかは値が変更可能かどうかと完全に一致するわけではありません) オブジェクトが mutable かどうかはその型によって決まります。例えば、数値型、文字列型とタプル型のインスタンスは immutable で、dict や list は mutable です。

オブジェクトを明示的に破壊することはできません; しかし、オブジェクトに到達不能 (unreachable) になると、ガベージコレクション (garbage-collection) によって処理されるかもしれません。ガベージコレクションを遅らせたり、全く行わない実装も許されています --- 到達可能なオブジェクトを処理してしまわないかぎり、ガベージコレクションをどう実装するかは実装品質の問題です。

[*1] 特定の条件が満たされた場合、オブジェクトの type を変更することが **できます** 。これは、正しく扱われなかった場合にとても奇妙な動作を引き起こすので、一般的には良い考えではありません。

CPython 実装の詳細: 現在の CPython 実装では参照カウント (reference-counting) 方式を使っており、(オプションとして) 循環参照を行っているガベージオブジェクトを遅延検出します。この実装ではほとんどのオブジェクトを到達不能になると同時に処理することができますが、循環参照を含むガベージオブジェクトの収集が確実に行われるよう保証しているわけではありません。循環参照を持つガベージオブジェクト収集の制御については、`gc` モジュールを参照してください。CPython 以外の実装は別の方式を使っており、CPython も将来は別の方式を使うかもしれません。オブジェクトが到達不能になったときに即座に終了処理されることに頼らないでください (ですからファイルは必ず明示的に閉じてください)。

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

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

他のオブジェクトに対する参照をもつオブジェクトもあります; これらは **コンテナ** *(container)* と呼ばれます。コンテナオブジェクトの例として、タプル、リスト、および辞書が挙げられます。オブジェクトへの参照自体がコンテナの値の一部です。ほとんどの場合、コンテナの値というと、コンテナに入っているオブジェクトの値のことを指し、それらオブジェクトのアイデンティティではありません; しかしながら、コンテナの変更可能性について述べる場合、今まさにコンテナに入っているオブジェクトのアイデンティティのことを指します。したがって、(タプルのように) 変更不能なオブジェクトが変更可能なオブジェクトへの参照を含む場合、その値が変化するのは変更可能なオブジェクトが変更された時、ということになります。

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. For example, after `a = 1; b = 1`, *a* and *b* may or may not refer to the same object with the value one, depending on the implementation. This is because `int` is an immutable type, so the reference to `1` can be reused. This behaviour depends on the implementation used, so should not be relied upon, but is something to be aware of when making use of object identity tests. However, after `c = []; d = []`, *c* and *d* are guaranteed to refer to two different, unique, newly created empty lists. (Note that `e = f = []` assigns the *same* object to both *e* and *f*.)

3.2 標準型の階層

以下は Python に組み込まれている型のリストです。(実装によって、C、Java、またはその他の言語で書かれた) 拡張モジュールで、その他の型が定義されていることがあります。新たな型 (有理数や、整数を効率的に記憶する配列、など) の追加は、たいてい標準ライブラリを通して提供されますが、将来のバージョンの Python では、型の階層構造にこのような追加がなされるかもしれません。

以下に説明する型のいくつかには、' 特殊属性 (special attribute)' を列挙した段落があります。これらの属性は実装へのアクセス手段を提供するもので、一般的な用途に利用するためのものではありません。特殊属性の定義は将来変更される可能性があります。

3.2.1 None

この型には単一の値しかありません。この値を持つオブジェクトはただ一つしか存在しません。このオブジェクトは組み込み名 `None` でアクセスされます。このオブジェクトは、様々な状況で値が存在しないことをしめします。例えば、明示的に値を返さない関数は `None` を返します。`None` の真値 (truth value) は偽 (false) です。

3.2.2 NotImplemented

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

詳細は implementing-the-arithmetic-operations を参照してください。

バージョン 3.9 で変更: Evaluating `NotImplemented` in a boolean context is deprecated. While it currently evaluates as true, it will emit a `DeprecationWarning`. It will raise a `TypeError` in a future version of Python.

3.2.3 Ellipsis

この型には単一の値しかありません。この値を持つオブジェクトはただ一つしか存在しません。このオブジェクトはリテラル `...` または Python で決められている名前 `Ellipsis` でアクセスされます。真理値は真 (true) です。

3.2.4 numbers.Number

数値リテラルによって作成されたり、算術演算や組み込みの算術関数によって返されるオブジェクトです。数値オブジェクトは変更不能です; 一度値が生成されると、二度と変更されることはありません。Python の数値オブジェクトはいうまでもなく数学で言うところの数値と強く関係していますが、コンピュータ内で数値を表現する際に伴う制限を受けています。

__repr__() と *__str__()* から計算された数値クラスの文字列表現には次のような特性があります:

- その文字列は、クラスコンストラクタに渡したときに、元の数値の値を持つオブジェクトを生成する有効な数値リテラルです。
- できるなら、10 を底として表現されます。
- 小数点の前にある 1 つのゼロを除いて、上に連なるゼロは表示されません。
- 小数点の後にある 1 つのゼロを除いて、下に連なるゼロは表示されません。
- 符号は数値が負数のときのみ表示されます。

Python distinguishes between integers, floating-point numbers, and complex numbers:

numbers.Integral (整数)

整数型は、整数 (正の数および負の数) を表す数学的集合内における要素を表現する型です。

注釈

整数表現に関する規則は、負の整数を含むシフト演算やマスク演算において、最も有意義な解釈ができるように意図されています。

整数には 2 種類あります:

整数 (`int`) 無
: 制限の範囲の数を表現しますが、利用可能な (仮想) メモリサイズの制限のみを受けます。シフト演算やマスク演算のために 2 進数表現を持つと想定されます。負の数は符号ビットが左に無限に延びているような錯覚を与える 2 の補数表現の変型で表されます。

ブール値 (bool) 真

偽値の False と True を表します。`False` と `True` を表す 2 つのオブジェクトのみがブール値オブジェクトです。ブール型は整数型の派生型であり、ほとんどの状況でそれぞれ 0 と 1 のように振る舞いますが、例外として文字列に変換されたときはそれぞれ `"False"` および `"True"` という文字列が返されます。

numbers.Real (float) (実数)

These represent machine-level double precision floating-point numbers. You are at the mercy of the underlying machine architecture (and C or Java implementation) for the accepted range and handling of overflow. Python does not support single-precision floating-point numbers; the savings in processor and memory usage that are usually the reason for using these are dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating-point numbers.

numbers.Complex (complex)

These represent complex numbers as a pair of machine-level double precision floating-point numbers. The same caveats apply as for floating-point numbers. The real and imaginary parts of a complex number `z` can be retrieved through the read-only attributes `z.real` and `z.imag`.

3.2.5 シーケンス型 (sequence)

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

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

シーケンスによっては、第三の ” ステップ (step)” パラメタを持つ ” 拡張スライス (extended slice)” もサポートしています: `a[i:j:k]` は、`x = i + n*k`, n `>= 0` かつ i `<=` x `<` j であるようなインデクス x を持つような a 全ての要素を選択します。

シーケンスは、変更可能なものか、そうでないかで区別されています:

変更不能なシーケンス (immutable sequence)

変更不能なシーケンス型のオブジェクトは、一度生成されるとその値を変更することができません。(オブジェクトに他のオブジェクトへの参照が入っている場合、参照されているオブジェクトは変更可能なオブジェクトでもよく、その値は変更される可能性があります; しかし、変更不能なオブジェクトが直接参照しているオブジェクトの集合自体は、変更することができません。)

以下の型は変更不能なシーケンス型です:

文字列型 (string) 文

字列は Unicode コードポイントを表現する値の配列です。文字列中のどのコードポイントも `U+0000 - U+10FFFF` の範囲で表現されることができます。Python は `char` 型を持ちません。代わりに、文字列中のどのコードポイントも長さ "1" の文字列オブジェクトとして表現することができます。組み込み関数 `ord()` は文字列形式を `U+0000 - U+10FFFF` の範囲の整数に変換します。また、組み込み関数 `chr()` は `0 - 10FFFF` の範囲の整数を対応する長さ `1` の文字列に変換します。`str.encode()` はテキストエンコーディングを使うことで `str` を `bytes` に変換するために使うことができます。また、`bytes.decode()` によりその逆が実行することができます。

タプル型 (tuple) タ

プルの要素は任意の Python オブジェクトです。二つ以上の要素からなるタプルは、個々の要素を表現する式をカンマで区切って構成します。単一の要素からなるタプル (単集合 'singleton') を作るには、要素を表現する式の直後にカンマをつけます (単一の式だけではタプルを形成しません。これは、式をグループ化するのに丸括弧を使えるようにしなければならないからです)。要素の全くない丸括弧の対を作ると空のタプルになります。

bytes A

bytes object is an immutable array. The items are 8-bit bytes, represented by integers in the range 0 <= x < 256. Bytes literals (like `b'abc'`) and the built-in `bytes()` constructor can be used to create bytes objects. Also, bytes objects can be decoded to strings via the `decode()` method.

変更可能なシーケンス型 (mutable sequence)

変更可能なシーケンスは、作成した後で変更することができます。変更可能なシーケンスでは、添字表記やスライス表記を使って指定された要素に代入を行うことができ、*`del`* (delete) 文を使って要素を削除することができます。

注釈

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

Python に最初から組み込まれている変更可能なシーケンス型は、今のところ二つです:

リスト型 (list) リ

ストの要素は任意の Python オブジェクトにできます。リストは、角括弧の中にカンマで区切られた式を並べて作ります。(長さが 0 や 1 のシーケンスを作るために特殊な場合分けは必要ないことに注意してください。)

バイト配列

bytearray オブジェクトは変更可能な配列です。組み込みの `bytearray()` コンストラクタによって作成されます。変更可能なことを除けば (つまりハッシュ化できない)、byte array は変更不能な `bytes` オブジェクトと同じインターフェースと機能を提供します。

3.2.6 集合型

集合型は、順序のない、ユニークで不変なオブジェクトの有限集合を表現します。そのため、(配列の) 添字を使ったインデックスアクセスはできません。ただし、イテレートは可能で、組み込み関数 `len()` は集合の要素数を返します。集合型の一般的な使い方は、集合に属しているかの高速なテスト、シーケンスからの重複の排除、共通集合・和集合・差・対称差といった数学的な演算の計算です。

集合の要素には、辞書のキーと同じ普遍性に関するルールが適用されます。数値型は通常の数値比較のルールに従うことに注意してください。もし 2 つの数値の比較結果が同値である (例えば、`1` と `1.0`) なら、そのうちの 1 つのみを集合に含めることができます。

現在、2 つの組み込み集合型があります:

集合型 可

変な集合型です。組み込みの `set()` コンストラクタで作成され、後から `add()` などのいくつかのメソッドで更新できます。

Frozen set **型** 不

変な集合型です。組み込みの `frozenset()` コンストラクタによって作成されます。frozenset は不変で **ハッシュ可能** なので、別の集合型の要素になったり、辞書のキーにすることができます。

3.2.7 マッピング型 (mapping)

任意のインデクス集合でインデクス化された、オブジェクトからなる有限の集合を表現します。添字表記 `a[k]` は、`k` でインデクス指定された要素を `a` から選択します; 選択された要素は式の中で使うことができ、代入や *del* 文の対象にすることができます。組み込み関数 `len()` は、マッピング内の要素数を返します。

Python に最初から組み込まれているマッピング型は、今のところ一つだけです:

辞書型 (dictionary)

ほぼ任意の値でインデクスされたオブジェクトからなる有限の集合を表します。キー (key) として使えない値の唯一の型は、リストや辞書、そしてオブジェクトの同一性でなく値で比較されるその他の変更可能な型です。これは、辞書型を効率的に実装する上で、キーのハッシュ値が不変である必要があるためです。数値型をキーに使う場合、キー値は通常の数値比較における規則に従います: 二つの値が等しくなる場合 (例えば `1` と `1.0`)、互いに同じ辞書のエントリを表すインデクスとして使うことができます。

辞書は挿入の順序を保持します。つまり、キーは辞書に追加された順番に生成されていきます。既存のキーを置き換えても、キーの順序は変わりません。キーを削除したのちに再挿入すると、元の場所ではなく辞書の最後に追加されます。

Dictionaries are mutable; they can be created by the `{}` notation (see section **辞書表示**).

拡張モジュール `dbm.ndbm` 、`dbm.gnu` は、`collections` モジュールのように、別のマッピング型の例を提供しています。

バージョン 3.7 で変更: Python のバージョン 3.6 では、辞書は挿入順序を保持しませんでした。CPython 3.6 では挿入順序は保持されましたが、それは策定された言語の仕様というより、その当時の実装の細部とみなされていました。

3.2.8 呼び出し可能型 (callable type)

関数呼び出し操作 (**呼び出し** *(call)* 参照) を行うことができる型です:

ユーザ定義関数 (user-defined function)

ユーザ定義関数オブジェクトは、関数定義を行うことで生成されます (**関数定義** 参照)。関数は、仮引数 (formal parameter) リストと同じ数の要素が入った引数リストとともに呼び出されます。

Special read-only attributes

属性	意味
`function.__globals__`	A reference to the `dictionary` that holds the function's *global variables* -- the global namespace of the module in which the function was defined.
`function.__closure__`	`None` or a `tuple` of cells that contain bindings for the function's free variables. セルオブジェクトは属性 `cell_contents` を持っています。これはセルの値を設定するのに加えて、セルの値を得るのにも使えます。

Special writable attributes

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

属性	意味
`function.__doc__`	The function's documentation string, or `None` if unavailable. Not inherited by subclasses.
`function.__name__`	The function's name. See also: `__name__ attributes`.
`function.__qualname__`	The function's *qualified name*. See also: `__qualname__ attributes`. Added in version 3.3.
`function.__module__`	関数が定義されているモジュールの名前です。モジュール名がない場合は `None` になります。
`function.__defaults__`	A `tuple` containing default *parameter* values for those parameters that have defaults, or `None` if no parameters have a default value.
`function.__code__`	The *code object* representing the compiled function body.
`function.__dict__`	The namespace supporting arbitrary function attributes. See also: `__dict__ attributes`.
`function.__annotations__`	A `dictionary` containing annotations of *parameters*. The keys of the dictionary are the parameter names, and `'return'` for the return annotation, if provided. See also: annotations-howto.
`function.__kwdefaults__`	A `dictionary` containing defaults for keyword-only *parameters*.
`function.__type_params__`	A `tuple` containing the *type parameters* of a *generic function*. Added in version 3.12.

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

CPython 実装の詳細: CPython's current implementation only supports function attributes on user-defined functions. Function attributes on *built-in functions* may be supported in the future.

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

インスタンスメソッド

インスタンスメソッドオブジェクトは、クラス、クラスインスタンスと任意の呼び出し可能オブジェクト (通常はユーザ定義関数) を結びつけます。

Special read-only attributes:

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

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

User-defined method objects may be created when getting an attribute of a class (perhaps via an instance of that class), if that attribute is a user-defined *function object* or a `classmethod` object.

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

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

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

When an instance method object is derived from a `classmethod` object, the "class instance" stored in *__self__* will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

It is important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

ジェネレータ関数 (generator function)

yield 文 (*yield* 文 の節を参照) を使う関数もしくはメソッドは **ジェネレータ関数** と呼ばれます。そのような関数が呼び出されたときは常に、関数の本体を実行するのに使える イテレータ オブジェクトを返します: イテレータの `iterator.__next__()` メソッドを呼び出すと、`yield` 文を使って値が提供されるまで関数を実行します。関数の *return* 文を実行するか終端に達したときは、`StopIteration` 例外が送出され、イテレータが返すべき値の最後まで到達しています。

コルーチン関数 (coroutine function)

async def を使用して定義された関数やメソッドを **コルーチン関数** *(coroutine function)* と呼びます。呼び出された時、そのような関数は *coroutine* オブジェクトを返します。コルーチン関数は *async with* や *async for* 文だけでなく *await* 式を持つことが出来ます。コルーチンオブジェクト を参照してください。

非同期ジェネレータ関数 (asynchronous generator function)

async def を使って定義され、*yield* 文を使用している関数やメソッドを *asynchronous generator function* と呼びます。そのような関数は、呼び出されたとき、非同期イテレータ オブジェクトを返します。このオブジェクトは *async for* 文で関数の本体を実行するのに使えます。

非同期イテレータの *aiterator.__anext__* メソッドを呼び出すと、他の処理が待たされているときに、*yield* 式を使い値を提供するところまで処理を進める *awaitable* を返します。その関数が空の *return* 文を実行する、もしくは処理の終わりに到達したときは、`StopAsyncIteration` 例外が送出され、非同期イテレータは出力すべき値の最後に到達したことになります。

組み込み関数 (built-in function)

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

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

組み込みメソッド (built-in method)

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

クラス

Classes are callable. These objects normally act as factories for new instances of themselves, but variations are possible for class types that override *__new__()*. The arguments of the call are passed to `__new__()` and, in the typical case, to *__init__()* to initialize the new instance.

クラスのインスタンス

任意のクラスのインスタンスは、クラスで *__call__()* メソッドを定義することで呼び出し可能になります。

3.2.9 モジュール

Modules are a basic organizational unit of Python code, and are created by the *import system* as invoked either by the *import* statement, or by calling functions such as `importlib.import_module()` and built-in `__import__()`. A module object has a namespace implemented by a `dictionary` object (this is the dictionary referenced by the *__globals__* attribute of functions defined in the module). Attribute references are translated to lookups in this dictionary, e.g., `m.x` is equivalent to `m.__dict__["x"]`. A module object does not contain the code object used to initialize the module (since it isn't needed once the initialization is done).

属性の代入を行うと、モジュールの名前空間辞書の内容を更新します。例えば、`m.x = 1` は `m.__dict__["x"] = 1` と同じです。

定義済みの (書き込み可能な) 属性:

__name__ モ
: ジュールの名前。

__doc__ モ
: ジュールのドキュメントで、文字列か、もし利用できない場合は None です。

__file__
: ロードされたモジュールファイルのパス名です。インタプリタに静的にリンクされている C モジュールのような特定の種類のモジュールでは、*__file__* 属性は存在しないかもしれません。共有ライブラリから動的にロードされた拡張モジュールの場合、この属性は 共有ライブラリファイルのパス名になります。

__annotations__ モ
: ジュールの本体の実行中に収集した **変数アノテーション** を格納する辞書です。__annotations__ を利用するベストプラクティスについては、アノテーションの HOWTO を参照してください。

読み出し専用の特殊属性: __dict__ はモジュールの名前空間で、辞書オブジェクトです。

CPython 実装の詳細: CPython がモジュール辞書を削除する方法により、モジュール辞書が生きた参照を持っていたとしてもその辞書はモジュールがスコープから外れた時に削除されます。これを避けるには、辞書をコピーするか、辞書を直接使っている間モジュールを保持してください。

3.2.10 カスタムクラス型

Custom class types are typically created by class definitions (see section **クラス定義**). A class has a namespace implemented by a dictionary object. Class attribute references are translated to lookups in this dictionary, e.g., C.x is translated to C.__dict__["x"] (although there are a number of hooks which allow for other means of locating attributes). When the attribute name is not found there, the attribute search continues in the base classes. This search of the base classes uses the C3 method resolution order which behaves correctly even in the presence of 'diamond' inheritance structures where there are multiple inheritance paths leading back to a common ancestor. Additional details on the C3 MRO used by Python can be found at python_2.3_mro.

When a class attribute reference (for class C, say) would yield a class method object, it is transformed into an instance method object whose *__self__* attribute is C. When it would yield a staticmethod object, it is transformed into the object wrapped by the static method object. See section **デスクリプタ *(descriptor)* の実装** for another way in which attributes retrieved from a class may differ from those actually contained in its __dict__.

クラス属性を代入すると、そのクラスの辞書だけが更新され、基底クラスの辞書は更新しません。

クラスオブジェクトを呼び出す (上記を参照) と、クラスインスタンスを生成します (下記を参照)。

特殊属性:

__name__ ク
ラス名。

__module__ ク
ラスが定義されているモジュールの名前。

__dict__ ク
ラスの名前空間を格納している辞書。

__bases__
ベースクラスリストに現れる順序でベースクラスを格納しているタプル。

__doc__ ク
ラスのドキュメントで、文字列か、もし未定義の場合は None です。

__annotations__ ク
ラスの本体の実行中に収集した **変数アノテーション** を格納する辞書です。__annotations__ を利用するベストプラクティスについては、アノテーションの HOWTO を参照してください。

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

__static_attributes__ A
tuple containing names of attributes of this class which are assigned through self.X from any function in its body.

__firstlineno__
The line number of the first line of the class definition, including decorators.

3.2.11 クラスインスタンス (class instance)

A class instance is created by calling a class object (see above). A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance's class has an attribute by that name, the search continues with the class attributes. If a class attribute is found that is a user-defined function object, it is transformed into an instance method object whose *__self__* attribute is the instance. Static method and class method objects are also transformed; see above under "Classes". See section **デスクリプタ** *(descriptor)* **の実装** for another way in which attributes of a class retrieved via its instances may differ from the objects actually stored in the class's __dict__. If no class attribute is found, and the object's class has a *__getattr__()* method, that is called to satisfy the lookup.

属性の代入や削除を行うと、インスタンスの辞書を更新しますが、クラスの辞書を更新することはありません。クラスで *__setattr__()* や *__delattr__()* メソッドが定義されている場合、直接インスタンスの辞書を更新す

る代わりにこれらのメソッドが呼び出されます。

クラスインスタンスは、ある特定の名前のメソッドを持っている場合、数値型やシーケンス型、あるいはマップ型のように振舞うことができます。特殊メソッド名 を参照してください。

特殊属性: `__dict__` は属性の辞書です; `__class__` はインスタンスのクラスです。

3.2.12 I/O オブジェクト (ファイルオブジェクトの別名)

file object は開かれたファイルを表します。ファイルオブジェクトを作るための様々なショートカットがあります: `open()` 組み込み関数、`os.popen()` 、`os.fdopen()` 、ソケットオブジェクトの `makefile()` メソッド (あるいは拡張モジュールから提供される他の関数やメソッド) 。

オブジェクト `sys.stdin` 、`sys.stdout` および `sys.stderr` は、インタプリタの標準入力、標準出力、および標準エラー出力ストリームに対応するファイルオブジェクトに初期化されます。これらはすべてテキストモードで開かれ、`io.TextIOBase` 抽象クラスによって定義されたインターフェースに従います。

3.2.13 内部型 (internal type)

インタプリタが内部的に使っているいくつかの型は、ユーザに公開されています。これらの定義は将来のインタプリタのバージョンでは変更される可能性がありますが、ここでは記述の完全性のために触れておきます。

コードオブジェクト

コードオブジェクトは **バイトコンパイルされた** *(byte-compiled)* 実行可能な Python コード、別名 バイトコード を表現します。コードオブジェクトと関数オブジェクトの違いは、関数オブジェクトが関数のグローバル変数 (関数を定義しているモジュールのグローバル) に対して明示的な参照を持っているのに対し、コードオブジェクトにはコンテキストがないということです; また、関数オブジェクトではデフォルト引数値を記憶できますが、コードオブジェクトではできません (実行時に計算される値を表現するため)。関数オブジェクトと違い、コードオブジェクトは変更不可能で、変更可能なオブジェクトへの参照を (直接、間接に関わらず) 含みません。

Special read-only attributes

codeobject.co_name	The function name
codeobject.co_qualname	The fully qualified function name Added in version 3.11.
codeobject.co_argcount	The total number of positional *parameters* (including positional-only parameters and parameters with default values) that the function has
codeobject.co_posonlyargcount	The number of positional-only *parameters* (including arguments with default values) that the function has
codeobject.co_kwonlyargcount	The number of keyword-only *parameters* (including arguments with default values) that the function has
codeobject.co_nlocals	The number of *local variables* used by the function (including parameters)
codeobject.co_varnames	A `tuple` containing the names of the local variables in the function (starting with the parameter names)
codeobject.co_cellvars	A `tuple` containing the names of *local variables* that are referenced by nested functions inside the function
codeobject.co_freevars	A `tuple` containing the names of free variables in the function
codeobject.co_code	A string representing the sequence of *bytecode* instructions in the function
codeobject.co_consts	A `tuple` containing the literals used by the *bytecode* in the function
codeobject.co_names	A `tuple` containing the names used by the *bytecode* in the function
codeobject.co_filename	The name of the file from which the code was compiled
codeobject.co_firstlineno	The line number of the first line of the function

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

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

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

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

Methods on code objects

`codeobject.co_positions()`

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

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

This positional information can be missing. A non-exhaustive lists of cases where this may happen:

- Running the interpreter with `-X no_debug_ranges`.
- Loading a pyc file compiled while using `-X no_debug_ranges`.
- Position tuples corresponding to artificial instructions.
- Line and column numbers that can't be represented due to implementation specific limitations.

When this occurs, some or all of the tuple elements can be `None`.

Added in version 3.11.

注釈

This feature requires storing column positions in code objects which may result in a small increase of disk usage of compiled Python files or interpreter memory usage. To avoid storing the extra information and/or deactivate printing the extra traceback information, the `-X no_debug_ranges` command line flag or the `PYTHONNODEBUGRANGES` environment variable can be used.

codeobject.co_lines()

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

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

The items yielded will have the following properties:

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

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

Added in version 3.10.

参考

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

codeobject.replace(***kwargs*)

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

Code objects are also supported by the generic function `copy.replace()`.

Added in version 3.8.

フレーム (frame) オブジェクト

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

Special read-only attributes

`frame.f_back`	Points to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame
`frame.f_code`	The *code object* being executed in this frame. Accessing this attribute raises an auditing event `object.__getattr__` with arguments `obj` and `"f_code"`.
`frame.f_locals`	The mapping used by the frame to look up *local variables*. If the frame refers to an *optimized scope*, this may return a write-through proxy object. バージョン 3.13 で変更: Return a proxy for optimized scopes.
`frame.f_globals`	The dictionary used by the frame to look up *global variables*
`frame.f_builtins`	The dictionary used by the frame to look up *built-in (intrinsic) names*
`frame.f_lasti`	The ”precise instruction” of the frame object (this is an index into the *bytecode* string of the *code object*)

Special writable attributes

`frame.f_trace`	If not `None`, this is a function called for various events during code execution (this is used by debuggers). Normally an event is triggered for each new source line (see *f_trace_lines*).
`frame.f_trace_lines`	Set this attribute to `False` to disable triggering a tracing event for each source line.
`frame.f_trace_opcodes`	Set this attribute to `True` to allow per-opcode events to be requested. Note that this may lead to undefined interpreter behaviour if exceptions raised by the trace function escape to the function being traced.
`frame.f_lineno`	The current line number of the frame -- writing to this from within a trace function jumps to the given line (only for the bottom-most frame). A debugger can implement a Jump command (aka Set Next Statement) by writing to this attribute.

Frame object methods

フレームオブジェクトはメソッドを一つサポートします:

`frame.clear()`

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

`RuntimeError` is raised if the frame is currently executing or suspended.

Added in version 3.4.

バージョン 3.13 で変更: Attempting to clear a suspended frame raises `RuntimeError` (as has always been the case for executing frames).

トレースバック (traceback) オブジェクト

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

バージョン 3.7 で変更: Traceback objects can now be explicitly instantiated from Python code.

For implicitly created tracebacks, when the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section *try* 文.) It is accessible as the third item of the tuple returned by `sys.exc_info()`, and as the `__traceback__` attribute of the caught exception.

When the program contains no suitable handler, the stack trace is written (nicely formatted) to the standard error stream; if the interpreter is interactive, it is also made available to the user as `sys.last_traceback`.

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

Special read-only attributes:

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

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

`traceback.tb_next`

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

バージョン 3.7 で変更: This attribute is now writable

スライス (slice) オブジェクト

スライスオブジェクトは、*__getitem__()* メソッドのためのスライスを表すのに使われます。スライスオブジェクトは組み込みの `slice()` 関数でも生成されます。

読み出し専用の特殊属性: `start` は下限です; `stop` は上限です; `step` はステップの値です; それぞれ省略された場合は `None` となっています。これらの属性は任意の型を持てます。

スライスオブジェクトはメソッドを一つサポートします:

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

このメソッドは単一の整数引数 *length* を取り、スライスオブジェクトが *length* 要素のシーケンスに適用されたときに表現する、スライスに関する情報を計算します。このメソッドは 3 つの整数からなるタプルを返します; それぞれ *start* および *stop* のインデックスと、*step* すなわちスライスのまたぎ幅です。インデックス値がないか、範囲外の値であれば、通常のスライスと変わらないやりかたで扱われます。

静的メソッド (static method) オブジェクト

静的メソッドは、上で説明したような関数オブジェクトからメソッドオブジェクトへの変換を阻止するための方法を提供します。静的メソッドオブジェクトは他の何らかのオブジェクト、通常はユーザ定義メソッドオブジェクトを包むラッパです。静的メソッドをクラスやクラスインスタンスから取得すると、実際に返されるオブジェクトはラップされたオブジェクトになり、それ以上は変換の対象にはなりません。静的メソッドオブジェクトは通常呼び出し可能なオブジェクトをラップしますが、静的オブジェクト自体は呼び出し可能です。静的オブジェクトは組み込みコンストラクタ `staticmethod()` で生成されます。

クラスメソッドオブジェクト

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

3.3 特殊メソッド名

クラスは、特殊な名前のメソッドを定義して、特殊な構文 (算術演算や添え字表記、スライス表記など) による特定の演算を実装できます。これは、Python の演算子オーバロード (*operator overloading*) へのアプローチです。これにより、クラスは言語の演算子に対する独自の振る舞いを定義できます。例えば、あるクラスが *__getitem__()* という名前のメソッドを定義しており、`x` がこのクラスのインスタンスであるとすると、`x[i]` は `type(x).__getitem__(x, i)` とほぼ等価です。特に注釈のない限り、適切なメソッドが定義されていないとき、このような演算を試みると例外 (たいていは `AttributeError` か `TypeError`) が送出されます。

特殊メソッドに `None` を設定することは、それに対応する演算が利用できないことを意味します。例えば、クラス

の *__iter__()* を None に設定した場合、そのクラスはイテラブルにはならず、そのインスタンスに対し iter() を呼び出すと (*__getitem__()* に処理が戻されずに) TypeError を送出します。[*2]

組み込み型をエミュレートするクラスを実装するときは、模範とされるオブジェクトにとって意味がある範囲に実装をとどめるのが重要です。例えば、あるシーケンスは個々の要素の取得はきちんと動くかもしれませんが、スライスの展開が意味をなさないかもしれません。(W3C のドキュメントオブジェクトモデルにある NodeList インターフェースがその一例です。)

3.3.1 基本的なカスタマイズ

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

クラス *cls* の新しいインスタンスを作るために呼び出されます。*__new__()* は静的メソッドで (このメソッドは特別扱いされているので、明示的に静的メソッドと宣言する必要はありません)、インスタンスを生成するよう要求されているクラスを第一引数にとります。残りの引数はオブジェクトのコンストラクタの式 (クラスの呼び出し文) に渡されます。*__new__()* の戻り値は新しいオブジェクトのインスタンス (通常は *cls* のインスタンス) でなければなりません。

典型的な実装では、クラスの新たなインスタンスを生成するときには super().__new__(cls[, ...]) に適切な引数を指定してスーパクラスの *__new__()* メソッドを呼び出し、新たに生成されたインスタンスに必要な変更を加えてから返します。

もし *__new__()* が オブジェクトの作成中に呼び出され、*cls* のインスタンスを返した場合には、__init__(self[, ...]) のようにして新しいインスタンスの *__init__()* が呼び出されます。このとき、*self* は新たに生成されたインスタンスで、残りの引数はオブジェクトコンストラクタに渡された引数と同じになります。

__new__() が *cls* のインスタンスを返さない場合、インスタンスの *__init__()* メソッドは呼び出されません。

__new__() の主な目的は、変更不能な型 (int, str, tuple など) のサブクラスでインスタンス生成をカスタマイズすることにあります。また、クラス生成をカスタマイズするために、カスタムのメタクラスでよくオーバーライドされます。

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

インスタンスが (*__new__()* によって) 生成された後、それが呼び出し元に返される前に呼び出されます。引数はクラスのコンストラクタ式に渡したものです。基底クラスとその派生クラスがともに *__init__()* メソッドを持つ場合、派生クラスの *__init__()* メソッドは基底クラスの *__init__()* メソッドを明示的に呼び出して、インスタンスの基底クラス部分が適切に初期化されること保証しなければなりません。例えば、super().__init__([args...]) 。

[*2] The *__hash__()*, *__iter__()*, *__reversed__()*, *__contains__()*, *__class_getitem__()* and __fspath__() methods have special handling for this. Others will still raise a TypeError, but may do so by relying on the behavior that None is not callable.

__new__() と *__init__()* は連携してオブジェクトを構成する (*__new__()* が作成し、*__init__()* がそれをカスタマイズする) ので、*__init__()* から非 None 値を返してはいけません; そうしてしまうと、実行時に TypeError が送出されてしまいます。

object.__del__(*self*)

インスタンスが破棄されるときに呼び出されます。これはファイナライザや (適切ではありませんが) デストラクタとも呼ばれます。基底クラスが *__del__()* メソッドを持っている場合は、派生クラスの *__del__()* メソッドは何であれ、基底クラスの *__del__()* メソッドを明示的に呼び出して、インスタンスの基底クラス部分をきちんと確実に削除しなければなりません。

__del__() メソッドが破棄しようとしているインスタンスへの新しい参照を作り、破棄を送らせることは (推奨されないものの) 可能です。これはオブジェクトの **復活** と呼ばれます。復活したオブジェクトが再度破棄される直前に *__del__()* が呼び出されるかどうかは実装依存です; 現在の *CPython* の実装では最初の一回しか呼び出されません。

It is not guaranteed that *__del__()* methods are called for objects that still exist when the interpreter exits. weakref.finalize provides a straightforward way to register a cleanup function to be called when an object is garbage collected.

注釈

del x は直接 x.__del__() を呼び出しません --- 前者は x の参照カウントを 1 つ減らし、後者は x の参照カウントが 0 まで落ちたときのみ呼び出されます。

CPython 実装の詳細: 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.

参考

gc モジュールのドキュメント。

警告

メソッド *__del__()* は不安定な状況で呼び出されるため、実行中に発生した例外は無視され、代わりに sys.stderr に警告が表示されます。特に:

- *__del__()* は、任意のコードが実行されているときに、任意のスレッドから呼び出せます。

__del__() で、ロックを取ったり、ブロックするリソースを呼び出したりする必要がある場合、*__del__()* の実行により中断されたコードにより、そのリソースが既に取得されていて、デッドロックが起きるかもしれません。

- *__del__()* は、インタプリタのシャットダウン中に実行できます。従って、(他のモジュールも含めた) アクセスする必要があるグローバル変数はすでに削除されているか、None に設定されているかもしれません。Python は、単一のアンダースコアで始まる名前のグローバルオブジェクトは、他のグローバル変数が削除される前にモジュールから削除されることを保証します; そのようなグローバル変数への他からの参照が存在しない場合、*__del__()* メソッドが呼ばれた時点で、インポートされたモジュールがまだ利用可能であることを保証するのに役立つかもしれません。

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

repr() 組み込み関数によって呼び出され、オブジェクトを表す「公式の (official)」文字列を計算します。可能なら、これは (適切な環境が与えられれば) 同じ値のオブジェクトを再生成するのに使える、有効な Python 式のようなものであるべきです。できないなら、<...some useful description...> 形式の文字列が返されるべきです。戻り値は文字列オブジェクトでなければなりません。クラスが *__repr__()* を定義していて *__str__()* は定義していなければ、そのクラスのインスタンスの「非公式の (informal)」文字列表現が要求されたときにも *__repr__()* が使われます。

この関数はデバッグの際によく用いられるので、たくさんの情報を含み、あいまいでないような表記にすることが重要です。

object.**__str__**(*self*)

オブジェクトの「非公式の (informal)」あるいは表示に適した文字列表現を計算するために、str(object) と組み込み関数 format(), print() によって呼ばれます。戻り値は string オブジェクトでなければなりません。

__str__() が有効な Python 表現を返すことが期待されないという点で、このメソッドは *object.__repr__()* とは異なります: より便利な、または簡潔な表現を使用することができます。

組み込み型 object によって定義されたデフォルト実装は、*object.__repr__()* を呼び出します。

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

bytes によって呼び出され、オブジェクトのバイト文字列表現を計算します。これは bytes オブジェクトを返すべきです。

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

format() 組み込み関数、さらには **フォーマット済み文字列リテラル** の評価、str.format() メソッドによって呼び出され、オブジェクトの " フォーマット化された (formatted)" 文字列表現を作ります。*format_spec* 引数は、必要なフォーマット化オプションの記述を含む文字列です。*format_spec* 引数の解釈は、*__format__()* を実装する型によりますが、ほとんどのクラスは組み込み型のいずれかにフォーマッ

ト化を委譲したり、同じようなフォーマット化オプション構文を使います。

標準のフォーマット構文の解説は、formatspec を参照してください。

戻り値は文字列オブジェクトでなければなりません。

バージョン 3.4 で変更: 空でない文字列が渡された場合 `object` 自身の ___format___ メソッドは `TypeError` を送出します。

バージョン 3.7 で変更: `object.__format__(x, '')` は `format(str(x), '')` ではなく `str(x)` と等価になりました。

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

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

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

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

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

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

これらはいわゆる ” 拡張比較 (rich comparison)” メソッドです。演算子シンボルとメソッド名の対応は以下の通りです: `x<y` は `x.__lt__(y)` を呼び出します; `x<=y` は `x.__le__(y)` を呼び出します; `x==y` は `x.__eq__(y)` を呼び出します; `x!=y` は `x.__ne__(y)` を呼び出します; `x>y` は `x.__gt__(y)` を呼び出します; `x>=y` は `x.__ge__(y)` を呼び出します。

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

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

カスタムの比較演算をサポートしていて、辞書のキーに使うことができる **ハッシュ可能** オブジェクトを作るときの重要な注意点について、*`__hash__()`* のドキュメント内に書かれているので参照してください。

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

or indirect subclass of the left operand's type, the reflected method of the right operand has priority, otherwise the left operand's method has priority. Virtual subclassing is not considered.

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

`object.__hash__`(*self*)

組み込みの `hash()` 関数や、`set`, `frozenset`, `dict` のようなハッシュを使ったコレクション型の要素に対する操作から呼び出されます。`__hash__()` メソッドは整数を返さなければなりません。このメソッドに必要な性質は、比較結果が等しいオブジェクトは同じハッシュ値を持つということです; オブジェクトを比較するときでも利用される要素をタプルに詰めてハッシュ値を計算することで、それぞれの要素のハッシュ値を混合することをおすすめします。

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

注釈

`hash()` はオブジェクト独自の *__hash__()* メソッドが返す値を `Py_ssize_t` のサイズに切り詰めます。これは 64-bit でビルドされていると 8 バイトで、32-bit でビルドされていると 4 バイトです。オブジェクトの *__hash__()* が異なる bit サイズのビルドでも可搬性が必要である場合は、必ず全てのサポートするビルドの bit 幅をチェックしてください。そうする簡単な方法は `python -c "import sys; print(sys.hash_info.width)"` を実行することです。

クラスが *__eq__()* メソッドを定義していないなら、*__hash__()* メソッドも定義してはなりません; クラスが *__eq__()* を定義していても *__hash__()* を定義していないなら、そのインスタンスはハッシュ可能コレクションの要素として使えません。クラスがミュータブルなオブジェクトを定義しており、*__eq__()* メソッドを実装しているなら、*__hash__()* を定義してはなりません。これは、**ハッシュ可能** コレクションの実装においてキーのハッシュ値がイミュータブルであることが要求されているからです (オブジェクトのハッシュ値が変化すると、誤ったハッシュバケツ: hash bucket に入ってしまいます)。

ユーザー定義クラスはデフォルトで *__eq__()* と *__hash__()* メソッドを持っています。このとき、(同一でない) すべてのオブジェクトは比較して異なり、`x.__hash__()` は `x == y` が `x is y` と `hash(x) == hash(y)` の両方を意味するような適切な値を返します。

__eq__() をオーバーライドしていて *__hash__()* を定義していないクラスでは、*__hash__()* は暗黙的に `None` に設定されます。クラスの *__hash__()* メソッドが `None` の場合、そのクラスのインスタンスのハッシュ値を取得しようとすると適切な `TypeError` が送出され、`isinstance(obj, collections.abc.Hashable)` でチェックするとハッシュ不能なものとして正しく認識されます。

__eq__() をオーバーライドしたクラスが親クラスからの *__hash__()* の 実装を保持したいなら、明示的に `__hash__ = <ParentClass>.__hash__` を設定することで、それをインタプリタに伝えなければなり

ません。

__eq__() をオーバーライドしていないクラスがハッシュサポートを抑制したい場合、クラス定義に `__hash__ = None` を含めてください。クラス自身で明示的に `TypeError` を送出する *__hash__()* を定義すると、`isinstance(obj, collections.abc.Hashable)` 呼び出しで誤ってハッシュ可能と識別されるでしょう。

注釈

デフォルトでは、文字列とバイト列の *__hash__()* 値は予測不可能なランダム値で "ソルト" されます。ハッシュ値は単独の Python プロセス内では定数であり続けますが、Python を繰り返し起動する毎に、予測できなくなります。

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

ハッシュ値の変更は、集合のイテレーション順序に影響します。Python はこの順序付けを保証していません (そして通常 32-bit と 64-bit の間でも異なります)。

`PYTHONHASHSEED` も参照してください。

バージョン 3.3 で変更: ハッシュのランダム化がデフォルトで有効になりました。

`object.__bool__`(*self*)

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

3.3.2 属性値アクセスをカスタマイズする

以下のメソッドを定義して、クラスインスタンスへの属性アクセス (`x.name` の使用、`x.name` への代入、`x.name` の削除) の意味をカスタマイズすることができます。

`object.__getattr__`(*self*, *name*)

デフォルトの属性アクセスが `AttributeError` で失敗したとき (*name* がインスタンスの属性または `self` のクラスツリーの属性でないために *__getattribute__()* が `AttributeError` を送出したか、*name* プロパティの *__get__()* が `AttributeError` を送出したとき) に呼び出されます。このメソッドは (計算された) 属性値を返すか、`AttributeError` 例外を送出しなければなりません。

Note that if the attribute is found through the normal mechanism, *__getattr__()* is not called. (This is an intentional asymmetry between *__getattr__()* and *__setattr__()*.) This is done both for

efficiency reasons and because otherwise *__getattr__()* would have no way to access other attributes of the instance. Note that at least for instance variables, you can take total control by not inserting any values in the instance attribute dictionary (but instead inserting them in another object). See the *__getattribute__()* method below for a way to actually get total control over attribute access.

object.**__getattribute__**(*self*, *name*)

クラスのインスタンスに対する属性アクセスを実装するために、無条件に呼び出されます。クラスが *__getattr__()* も定義している場合、*__getattr__()* は、*__getattribute__()* で明示的に呼び出すか、`AttributeError` 例外を送出しない限り呼ばれません。このメソッドは (計算された) 属性値を返すか、`AttributeError` 例外を送出します。このメソッドが再帰的に際限なく呼び出されてしまうのを防ぐため、実装の際には常に、必要な属性全てへのアクセスで、例えば `object.__getattribute__(self, name)` のように基底クラスのメソッドを同じ属性名を使って呼び出さなければなりません。

注釈

This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or *built-in functions*. See **特殊メソッド検索**.

セキュリティに関わるような `obj` と `name` を渡しての `object.__getattr__` を使った属性アクセスは 監査イベント を送出します。

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

属性の代入が試みられた際に呼び出されます。これは通常の代入の過程 (すなわち、インスタンス辞書への値の代入) の代わりに呼び出されます。*name* は属性名で、*value* はその属性に代入する値です。

__setattr__() の中でインスタンス属性への代入が必要なら、基底クラスのこれと同じ名前のメソッドを呼び出さなければなりません。例えば、`object.__setattr__(self, name, value)` とします。

セキュリティに関わるような `obj` と `name` と `value` を渡しての `object.__setattr__` を使った属性のアサインは 監査イベント を送出します。

object.**__delattr__**(*self*, *name*)

__setattr__() に似ていますが、代入ではなく値の削除を行います。このメソッドを実装するのは、オブジェクトにとって `del obj.name` が意味がある場合だけにしなければなりません。

セキュリティに関わるような `obj` と `name` を渡しての `object.__getattr__` を使った属性の削除は 監査イベント を送出します。

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

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

モジュールの属性値アクセスをカスタマイズする

特殊な名前の `__getattr__` と `__dir__` も、モジュール属性へのアクセスをカスタマイズするのに使えます。モジュールレベルの `__getattr__` 関数は属性名である 1 引数を受け取り、計算した値を返すか `AttributeError` を送出します。属性がモジュールオブジェクトから、通常の検索、つまり *object.__getattribute__()* で見付からなかった場合は、`AttributeError` を送出する前に、モジュールの `__dict__` から `__getattr__` が検索されます。見付かった場合は、その属性名で呼び出され、結果が返されます。

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

より細かい粒度でのモジュールの動作 (属性やプロパティの設定など) のカスタマイズのために、モジュールオブジェクトの `__class__` 属性に `types.ModuleType` のサブクラスが設定できます。例えば次のようになります:

```
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
```

注釈

モジュールの `__getattr__` を定義したり `__class__` を設定したりしても、影響があるのは属性アクセスの構文が使われる検索だけです -- モジュールの globals への直接アクセスは (モジュール内のコードからとモジュールの globals のどちらでも) 影響を受けません。

バージョン 3.5 で変更: モジュールの属性 `__class__` が書き込み可能になりました。

Added in version 3.7: `__getattr__` モジュール属性と `__dir__` モジュール属性。

参考

PEP 562 - **モジュールの __getattr__ と __dir__** モジュールの `__getattr__` 関数および `__dir__` 関数の説明。

デスクリプタ (descriptor) の実装

以下のメソッドは、このメソッドを持つクラス (いわゆる **デスクリプタ** *(descriptor)* クラス) のインスタンスが、**オーナー** *(owner)* クラスに存在するときにのみ適用されます (デスクリプタは、オーナーのクラス辞書か、その親のいずれかのクラス辞書になければなりません)。以下の例では、" 属性" とは、名前がオーナークラスの `__dict__` のプロパティ (porperty) のキーであるような属性を指します。

`object.__get__`(*self*, *instance*, *owner=None*)

オーナークラス（クラス属性アクセスの場合）や、クラスのインスタンス（インスタンス属性アクセスの場合）の属性取得時に呼び出されます。*instance* を通じて属性をアクセスする時に、オプションの *owner* 引数はオーナークラスです。*owner* を通じて属性アクセスするときは `None` です。

このメソッドは、算出された属性値を返すか、`AttributeError` 例外を送出します。

PEP 252 は *`__get__()`* は 1 つや 2 つの引数を持つ呼び出し可能オブジェクトであると定義しています。Python の組み込みのデスクリプタはこの仕様をサポートしていますが、サードパーティ製のツールの中には両方の引数を必要とするものもあります。Python の *`__getattribute__()`* 実装は必要かどうかに関わらず、両方の引数を常に渡します。

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

オーナークラスのインスタンス *instance* 上の属性を新たな値 *value* に設定する際に呼び出されます。

`__set__()` あるいは *`__delete__()`* を追加すると、デスクリプタは「データデスクリプタ」に変わります。詳細は **デスクリプタの呼び出し** を参照してください。

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

オーナークラスのインスタンス *instance* 上の属性を削除する際に呼び出されます。

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

`object.__objclass__`

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

デスクリプタの呼び出し

一般にデスクリプタとは、特殊な " 束縛に関する動作 (binding behaviour)" をもつオブジェクト属性のことです。デスクリプタは、デスクリプタプロトコル (descriptor protocol) のメソッド: *__get__()*, *__set__()*, および *__delete__()* を使って、属性アクセスをオーバーライドしているものです。これらのメソッドのいずれかがオブジェクトに対して定義されている場合、オブジェクトはデスクリプタであるといいます。

属性アクセスのデフォルトの動作は、オブジェクトの辞書から値を取り出したり、値を設定したり、削除したりするというものです。例えば、`a.x` による属性の検索では、まず `a.__dict__['x']`、次に `type(a).__dict__['x']`、そして `type(a)` の基底クラスでメタクラスでないものに続く、といった具合に連鎖が起こります。

しかし、検索対象の値が、デスクリプタメソッドのいずれかを定義しているオブジェクトであれば、Python はデフォルトの動作をオーバーライドして、代わりにデスクリプタメソッドを呼び出します。先述の連鎖の中のどこでデスクリプタメソッドが呼び出されるかは、どのデスクリプタメソッドが定義されていて、どのように呼び出されたかに依存します。

デスクリプタ呼び出しの基点となるのは、属性名への束縛 (binding)、すなわち `a.x` です。引数がどのようにデスクリプタに結合されるかは `a` に依存します:

直接呼び出し (Direct Call)
: 最も単純で、かつめったに使われない呼び出し操作は、コード中で直接デスクリプタメソッドの呼び出し: `x.__get__(a)` を行うというものです。

インスタンス束縛 (Instance Binding)
: オブジェクトインスタンスへ束縛すると、`a.x` は呼び出し `type(a).__dict__['x'].__get__(a, type(a))` に変換されます。

クラス束縛 (Class Binding)
: クラスへ束縛すると、`A.x` は呼び出し `A.__dict__['x'].__get__(None, A)` に変換されます。

super **束縛** (Super Binding)
: `super(A, a).x` のようなドットを使ったルックアップは `a.__class__.__mro__` を探索して、`A` の前のクラス `B` をまず探し、`B.__dict__['x'].__get__(a, A)` を返します。もしデスクリプタでなければ `x` を変更せずに返します。

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

(`@staticmethod` や `@classmethod` を含む) Python メソッドは、非データデスクリプタとして実装されていま

す。その結果、インスタンスではメソッドを再定義したりオーバーライドできます。このことにより、個々のインスタンスが同じクラスの他のインスタンスと互いに異なる動作を獲得することができます。

`property()` 関数はデータデスクリプタとして実装されています。従って、インスタンスはあるプロパティの動作をオーバーライドすることができません。

___slots___

___slots___ を使うと、(プロパティのように) データメンバを明示的に宣言し、(明示的に *___slots___* で宣言しているか親クラスに存在しているかでない限り) `__dict__` や *___weakref___* を作成しないようにできます。

`__dict__` を使うのに比べて、節約できるメモリ空間はかなり大きいです。属性探索のスピードもかなり向上できます。

`object.__slots__`

このクラス変数には、インスタンスが用いる変数名を表す、文字列、イテラブル、または文字列のシーケンスを代入できます。*___slots___* は、各インスタンスに対して宣言された変数に必要な記憶領域を確保し、`__dict__` と *___weakref___* が自動的に生成されないようにします。

Notes on using *___slots___*:

- *___slots___* を持たないクラスから継承するとき、インスタンスの `__dict__` 属性と *___weakref___* 属性は常に利用可能です。
- `__dict__` 変数がない場合、*___slots___* に列挙されていない新たな変数をインスタンスに代入することはできません。列挙されていない変数名を使って代入しようとした場合、`AttributeError` が送出されます。新たな変数を動的に代入したいのなら、*___slots___* を宣言する際に `'__dict__'` を変数名のシーケンスに追加してください。
- *___slots___* を定義しているクラスの各インスタンスに *___weakref___* 変数がない場合、インスタンスに対する弱参照 (`weak references`) はサポートされません。弱参照のサポートが必要なら、*___slots___* を宣言する際に `'__weakref__'` を変数名のシーケンスに追加してください。
- *___slots___* は、クラスのレベルで各変数に対する **デスクリプタ** を使って実装されます。その結果、*___slots___* に定義されているインスタンス変数のデフォルト値はクラス属性を使って設定できなくなっています; そうしないと、デスクリプタによる代入をクラス属性が上書きしてしまうからです。
- *___slots___* の宣言の作用は、それが定義されたクラスだけには留まりません。親クラスで宣言された *___slots___* は子クラスでも利用可能です。ただし、子クラスは、自身も *___slots___* (ここには **追加の** スロットの名前のみ含めるべき) を定義しない限り `__dict__` や *___weakref___* を持ちます。
- あるクラスで、基底クラスですでに定義されているスロットを定義した場合、基底クラスのスロットで定義されているインスタンス変数は (デスクリプタを基底クラスから直接取得しない限り) アクセスできなくなります。これにより、プログラムの趣意が不定になってしまいます。将来は、この問題を避けるために何らかのチェックが追加されるかもしれません。

- `TypeError` will be raised if nonempty *__slots__* are defined for a class derived from a `"variable-length" built-in type` such as `int`, `bytes`, and `tuple`.
- Any non-string *iterable* may be assigned to *__slots__*.
- If a `dictionary` is used to assign *__slots__*, the dictionary keys will be used as the slot names. The values of the dictionary can be used to provide per-attribute docstrings that will be recognised by `inspect.getdoc()` and displayed in the output of `help()`.
- `__class__` への代入は、両方のクラスが同じ *__slots__* を持っているときのみ動作します。
- 複数のスロットを持つ親クラスを使った 多重継承 はできますが、スロットで作成された属性を持つ親クラスは 1 つに限られます (他の基底クラスのスロットは空でなければなりません) - それに違反すると `TypeError` が送出されます。
- もし *__slots__* に対して **イテレータ** を使用すると、イテレータの値ごとに **デスクリプタ** が作られます。しかし、*__slots__* 属性は空のイテレータとなります。

3.3.3 クラス生成をカスタマイズする

クラスが他のクラスを継承するときに必ず、親クラスの *__init_subclass__()* が呼び出されます。これを利用すると、サブクラスの挙動を変更するクラスを書くことができます。これは、クラスデコレータととても良く似ていますが、クラスデコレータが、それが適用された特定のクラスにのみに影響するのに対して、`__init_subclass__` は、もっぱら、このメソッドを定義したクラスの将来のサブクラスに適用されます。

`classmethod` object.__init_subclass__(*cls*)

このメソッドは、それが定義されたクラスが継承された際に必ず呼び出されます。*cls* は新しいサブクラスです。もし、このメソッドがインスタンスメソッドとして定義されると、暗黙的にクラスメソッドに変換されます。

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

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

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

`object.__init_subclass__` のデフォルト実装は何も行いませんが、何らかの引数とともに呼び出された場合は、エラーを送出します。

注釈

メタクラスのヒント metaclass は残りの型機構によって消費され、__init_subclass__ 実装に渡されることはありません。実際のメタクラス (明示的なヒントではなく) は、type(cls) としてアクセスできます。

Added in version 3.6.

When a class is created, type.__new__() scans the class variables and makes callbacks to those with a *__set_name__()* hook.

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

オーナーとなるクラス *owner* が作成された時点で自動的に呼び出されます。オブジェクトはそのクラスの *name* に割り当てられます。

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

If the class variable is assigned after the class is created, *__set_name__()* will not be called automatically. If needed, *__set_name__()* can be called directly:

```
class A:
   pass

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

詳細は **クラスオブジェクトの作成** を参照してください。

Added in version 3.6.

メタクラス

デフォルトでは、クラスは type() を使って構築されます。クラス本体は新しい名前空間で実行され、クラス名が type(name, bases, namespace) の結果にローカルに束縛されます。

クラス生成プロセスはカスタマイズできます。そのためにはクラス定義行で metaclass キーワード引数を渡すか、そのような引数を定義行に含む既存のクラスを継承します。次の例で MyClass と MySubclass は両方とも Meta のインスタンスです:

```
class Meta(type):
    pass
```

(次のページに続く)

(前のページからの続き)

```
class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

クラス定義の中で指定された他のキーワード引数は、後述するすべてのメタクラス操作に渡されます。

クラス定義が実行される際に、以下のステップが生じます:

- MRO エントリの解決が行われる;
- 適切なメタクラスが決定される;
- クラスの名前空間が準備される;
- クラスの本体が実行される;
- クラスオブジェクトが作られる。

MRO エントリの解決

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

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

参考

`types.resolve_bases()`

Dynamically resolve bases that are not instances of `type`.

`types.get_original_bases()`

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

PEP 560

Core support for typing module and generic types.

適切なメタクラスの決定

クラス定義に対して適切なメタクラスは、以下のように決定されます:

- 基底も明示的なメタクラスも与えられていない場合は、type() が使われます;
- 明示的なメタクラスが与えられていて、それが type() のインスタンス **ではない** 場合、それをメタクラスとして直接使います;
- 明示的なメタクラスとして type() のインスタンスが与えられたか、基底が定義されていた場合は、最も派生した (継承関係で最も下の) メタクラスが使われます。

最も派生的なメタクラスは、(もしあれば) 明示的に指定されたメタクラスと、指定されたすべてのベースクラスのメタクラスから選ばれます。最も派生的なメタクラスは、これらのメタクラス候補のすべてのサブタイプであるようなものです。メタクラス候補のどれもその基準を満たさなければ、クラス定義は TypeError で失敗します。

クラスの名前空間の準備

適切なメタクラスが指定されると、クラスの名前空間が用意されます。もしメタクラスが __prepare__ 属性を持っている場合、namespace = metaclass.__prepare__(name, bases, **kwds) が呼ばれます。追加のキーワード引数は、もしクラス定義にあれば設定されます。__prepare__ メソッドは **クラスメソッド** として実装する必要があります。__prepare__ が作成して返した名前空間は __new__ に渡されますが、最終的なクラスオブジェクトは新しい dict にコピーして作成されます。

メタクラスに __prepare__ 属性がない場合、クラスの名前空間は空の 順序付きマッピングとして初期化されます。

参考

PEP 3115 - Metaclasses in Python 3000
: __prepare__ 名前空間フックの導入

クラス本体の実行

クラス本体が (大まかには) exec(body, globals(), namespace) として実行されます。通常の呼び出しと exec() の重要な違いは、クラス定義が関数内部で行われる場合、レキシカルスコープによってクラス本体 (任意のメソッドを含む) が現在のスコープと外側のスコープから名前を参照できるという点です。

しかし、クラス定義が関数内部で行われる時でさえ、クラス内部で定義されたメソッドはクラススコープで定義された名前を見ることはできません。クラス変数はインスタンスメソッドかクラスメソッドの最初のパラメータからアクセスするか、次の節で説明する、暗黙的に静的スコープが切られている __class__ 参照からアクセスしなければなりません。

クラスオブジェクトの作成

クラス本体の実行によってクラスの名前空間が初期化されたら、`metaclass(name, bases, namespace, **kwds)` を呼び出すことでクラスオブジェクトが作成されます (ここで渡される追加のキーワードは `__prepare__` に渡されるものと同じです)。

このクラスオブジェクトは、`super()` の無引数形式によって参照されるものです。`__class__` は、クラス本体中のメソッドが `__class__` または `super` のいずれかを参照している場合に、コンパイラによって作成される暗黙のクロージャー参照です。これは、メソッドに渡された最初の引数に基づいて現在の呼び出しを行うために使用されるクラスまたはインスタンスが識別される一方、`super()` の無引数形式がレキシカルスコープに基づいて定義されているクラスを正確に識別することを可能にします。

CPython 実装の詳細: CPython 3.6 以降では、`__class__` セルは、クラス名前空間にある `__classcell__` エントリーとしてメタクラスに渡されます。`__class__` セルが存在していた場合は、そのクラスが正しく初期化されるために、`type.__new__` の呼び出しに到達するまで上に伝搬されます。失敗した場合は、Python 3.8 では `RuntimeError` になります。

デフォルトのメタクラス `type` や最終的には `type.__new__` を呼び出すメタクラスを使っているときは、クラスオブジェクトを作成した後に次のカスタム化の手順が起動されます:

1) `type.__new__` メソッドが *`__set_name__()`* が定義されているクラスの名前空間にある全ての属性を収集します;
2) それらの `__set_name__` メソッドが、そのメソッドが定義されているクラス、およびそこに属する属性に割り当てられている名前を引数として呼び出されます;
3) 新しいクラスのメソッド解決順序ですぐ上に位置する親クラスで *`__init_subclass__()`* フックが呼び出されます。

クラスオブジェクトが作成された後には、クラス定義に含まれているクラスデコレータ (もしあれば) にクラスオブジェクトが渡され、デコレータが返すオブジェクトがここで定義されたクラスとしてローカルの名前空間に束縛されます。

新しいクラスが `type.__new__` で生成されたときは、名前空間引数として与えられたオブジェクトは新しい順序付きのマッピングに複製され、元のオブジェクトは破棄されます。新しく複製したものは読み出し専用のプロキシでラップされ、クラスオブジェクトの `__dict__` 属性になります。

参考

PEP 3135 - New super
: 暗黙の `__class__` クロージャ参照について記述しています

メタクラスの用途

メタクラスは限りない潜在的利用価値を持っています。これまで試されてきたアイデアには、列挙型、ログ記録、インターフェースのチェック、自動デリゲーション、自動プロパティ生成、プロキシ、フレームワーク、そして自動リソースロック／同期といったものがあります。

3.3.4 インスタンスのカスタマイズとサブクラスチェック

以下のメソッドは組み込み関数 `isinstance()` と `issubclass()` のデフォルトの動作を上書きするのに利用します。

特に、`abc.ABCMeta` メタクラスは、抽象基底クラス (ABCs) を" 仮想基底クラス (virtual base classes)" として、他の ABC を含む、任意のクラスや (組み込み型を含む) 型に追加するために、これらのメソッドを実装しています。

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

instance が (直接、または間接的に) *class* のインスタンスと考えられる場合に true を返します。定義されていれば、`isinstance(instance, class)` の実装のために呼び出されます。

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

subclass が (直接、または間接的に) *class* のサブクラスと考えられる場合に true を返します。定義されていれば、`issubclass(subclass, class)` の実装のために呼び出されます。

なお、これらのメソッドは、クラスの型 (メタクラス) 上で検索されます。実際のクラスにクラスメソッドとして定義することはできません。これは、インスタンスそれ自体がクラスであるこの場合にのみ、インスタンスに呼び出される特殊メソッドの検索と一貫しています。

参考

PEP 3119 - 抽象基底クラスの導入 抽象基底クラス (abc モジュールを参照) を言語に追加する文脈においての動機から、*__instancecheck__()* と *__subclasscheck__()* を通して、`isinstance()` と `issubclass()` に独自の動作をさせるための仕様の記述があります。

3.3.5 ジェネリック型をエミュレートする

When using *type annotations*, it is often useful to *parameterize* a *generic type* using Python's square-brackets notation. For example, the annotation `list[int]` might be used to signify a `list` in which all the elements are of type `int`.

参考

PEP 484 - 型ヒント
: Introducing Python's framework for type annotations

Generic Alias Types
: Documentation for objects representing parameterized generic classes

Generics, user-defined generics and `typing.Generic`
: 実行時にパラメータ設定が可能であり、かつ静的な型チェッカーが理解できるジェネリッククラスを実装する方法のドキュメントです。

A class can *generally* only be parameterized if it defines the special class method `__class_getitem__()`.

`classmethod` `object.__class_getitem__`(*cls*, *key*)
: *key* にある型引数で特殊化されたジェネリッククラスを表すオブジェクトを返します。

  When defined on a class, `__class_getitem__()` is automatically a class method. As such, there is no need for it to be decorated with `@classmethod` when it is defined.

The purpose of *__class_getitem__*

The purpose of *`__class_getitem__()`* is to allow runtime parameterization of standard-library generic classes in order to more easily apply *type hints* to these classes.

To implement custom generic classes that can be parameterized at runtime and understood by static type-checkers, users should either inherit from a standard library class that already implements *`__class_getitem__()`*, or inherit from `typing.Generic`, which has its own implementation of `__class_getitem__()`.

Custom implementations of *`__class_getitem__()`* on classes defined outside of the standard library may not be understood by third-party type-checkers such as mypy. Using `__class_getitem__()` on any class for purposes other than type hinting is discouraged.

__class_getitem__ versus *__getitem__*

Usually, the *subscription* of an object using square brackets will call the *__getitem__()* instance method defined on the object's class. However, if the object being subscribed is itself a class, the class method *__class_getitem__()* may be called instead. `__class_getitem__()` should return a GenericAlias object if it is properly defined.

Presented with the *expression* `obj[x]`, the Python interpreter follows something like the following process to decide whether *__getitem__()* or *__class_getitem__()* should be called:

```
from inspect import isclass

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

    class_of_obj = type(obj)

    # If the class of obj defines __getitem__,
    # call class_of_obj.__getitem__(obj, x)
    if hasattr(class_of_obj, '__getitem__'):
        return class_of_obj.__getitem__(obj, x)

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

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

In Python, all classes are themselves instances of other classes. The class of a class is known as that class's *metaclass*, and most classes have the `type` class as their metaclass. `type` does not define *__getitem__()*, meaning that expressions such as `list[int]`, `dict[str, float]` and `tuple[str, bytes]` all result in *__class_getitem__()* being called:

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

(次のページに続く)

(前のページからの続き)

```
>>> type(list[int])
<class 'types.GenericAlias'>
```

However, if a class has a custom metaclass that defines *__getitem__()*, subscribing the class may result in different behaviour. An example of this can be found in the `enum` module:

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

参考

PEP 560 - typing モジュールとジェネリック型に対する言語コアによるサポート
Introducing *__class_getitem__()*, and outlining when a *subscription* results in `__class_getitem__()` being called instead of *__getitem__()*

3.3.6 呼び出し可能オブジェクトをエミュレートする

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

インスタンスが関数として ” 呼ばれた” 際に呼び出されます。このメソッドが定義されている場合、`x(arg1, arg2, ...)` は大まかには `type(x).__call__(x, arg1, ...)` に変換されます。

3.3.7 コンテナをエミュレートする

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

`object.__len__`(*self*)

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

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

`object.__length_hint__`(*self*)

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

Added in version 3.4.

注釈

スライシングは、以下の 3 メソッドによって排他的に行われます。次のような呼び出しは

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

次のように翻訳され

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

以下も同様です。存在しないスライスの要素は `None` で埋められます。

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

Called to implement evaluation of `self[key]`. For *sequence* types, the accepted keys should be integers. Optionally, they may support `slice` objects as well. Negative index support is also optional. If *key* is of an inappropriate type, `TypeError` may be raised; if *key* is a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. For *mapping* types, if *key* is missing (not in the container), `KeyError` should be raised.

注釈

for ループでは、シーケンスの終端を正しく検出できるようにするために、不正なインデクスに対して `IndexError` が送出されるものと期待しています。

注釈

When *subscripting* a *class*, the special class method *__class_getitem__()* may be called instead of `__getitem__()`. See *__class_getitem__ versus __getitem__* for more details.

`object.__setitem__`(*self*, *key*, *value*)

`self[key]` に対する代入を実装するために呼び出されます。*__getitem__()* と同じ注意事項があてはまります。このメソッドを実装できるのは、あるキーに対する値の変更をサポートしているか、新たなキーを追加できるようなマップの場合と、ある要素を置き換えることができるシーケンスの場合だけです。不正な *key* に対しては、*__getitem__()* メソッドと同様の例外の送出を行わなければなりません。

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

`self[key]` の削除を実装するために呼び出されます。*__getitem__()* と同じ注意事項があてはまります。このメソッドを実装できるのは、キーの削除をサポートしているマップの場合と、要素を削除できるシーケンスの場合だけです。不正な *key* に対しては、*__getitem__()* メソッドと同様の例外の送出を行わなければなりません。

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

`self[key]` の実装において辞書内にキーが存在しなかった場合に、dict のサブクラスのために `dict.`*__getitem__()* によって呼び出されます。

object.__iter__(*self*)

このメソッドは、コンテナに対して **イテレータ** が要求された際に呼び出されます。このメソッドは、コンテナ内の全てのオブジェクトに渡って反復処理できるような、新たなイテレータオブジェクトを返さなければなりません。マッピングでは、コンテナ内のキーに渡って反復処理しなければなりません。

object.__reversed__(*self*)

reversed() 組み込み関数が逆方向イテレーションを実装するために、(存在すれば) 呼び出します。コンテナ内の全要素を逆順にイテレートする、新しいイテレータを返すべきです。

__reversed__() メソッドが定義されていない場合、reversed() 組込み関数は sequence プロトコル (*__len__()* と *__getitem__()*) を使った方法にフォールバックします。sequence プロトコルをサポートしたオブジェクトは、reversed() よりも効率のいい実装を提供できる場合にのみ *__reversed__()* を定義するべきです。

帰属テスト演算子 (*in* および *not in*) は通常、コンテナの要素に対する反復処理のように実装されます。しかし、コンテナオブジェクトで以下の特殊メソッドを定義して、より効率的な実装を行ったり、オブジェクトがイテラブルでなくてもよいようにできます。

object.__contains__(*self*, *item*)

帰属テスト演算を実装するために呼び出されます。*item* が *self* 内に存在する場合には真を、そうでない場合には偽を返さなければなりません。マップオブジェクトの場合、値やキーと値の組ではなく、キーに対する帰属テストを考えなければなりません。

__contains__() を定義しないオブジェクトに対しては、メンバシップテストはまず、*__iter__()* を使った反復を試みます、次に古いシーケンス反復プロトコル *__getitem__()* を使います、**言語レファレンスのこの節** を参照して下さい。

3.3.8 数値型をエミュレートする

以下のメソッドを定義して、数値型オブジェクトをエミュレートすることができます。特定の種類の数値型ではサポートされていないような演算に対応するメソッド (非整数の数値に対するビット単位演算など) は、未定義のままにしておかなければなりません。

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*)

これらのメソッドを呼んで二項算術演算子 (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |) を実装します。例えば *x* が *__add__()* メソッドのあるクラスのインスタンスである場合、式 x + y を評価すると type(x).__add__(x, y) が呼ばれます。*__divmod__()* メソッドは *__floordiv__()* と *__mod__()* を使用するのと等価でなければなりません。*__truediv__()* と関連してはなりません。組み込みの pow() 関数の三項のものがサポートされていなければならない場合、*__pow__()* はオプションの第三引数を受け取るものとして定義されなければなりません。

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

object.__radd__(*self*, *other*)

object.__rsub__(*self*, *other*)

object.__rmul__(*self*, *other*)

object.__rmatmul__(*self*, *other*)

object.__rtruediv__(*self*, *other*)

object.__rfloordiv__(*self*, *other*)

object.__rmod__(*self*, *other*)

object.__rdivmod__(*self*, *other*)

object.__rpow__(*self*, *other*[, *modulo*])

object.__rlshift__(*self*, *other*)

object.__rrshift__(*self*, *other*)

object.__rand__(*self*, *other*)

object.__rxor__(*self*, *other*)

object.__ror__(*self*, *other*)

These methods are called to implement the binary arithmetic operations (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation[*3] and the operands are of different types.[*4] For

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

[*4] 同じ型の被演算子については、無反転のメソッド (たとえば *__add__()*) が失敗した場合、その演算はサポートされていないとみなされます。これは、反射したメソッドが呼び出されない理由です。

instance, to evaluate the expression `x - y`, where *y* is an instance of a class that has an *__rsub__()* method, `type(y).__rsub__(y, x)` is called if `type(x).__sub__(x, y)` returns `NotImplemented`.

ただし、三項演算子 `pow()` が *__rpow__()* を呼ぶことはないので注意してください (型強制の規則が非常に難解になるからです)。

注釈

右側の被演算子の型が左側の被演算子の型のサブクラスであり、このサブクラスであるメソッドに対する反射メソッドと異なる実装が定義されている場合には、左側の被演算子の非反射メソッドが呼ばれる前に、このメソッドが呼ばれます。この振る舞いにより、サブクラスが親の演算をオーバーライドすることが可能になります。

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

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

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

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

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

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

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

`object.__ipow__`(*self*, *other*[, *modulo*])

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

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

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

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

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

These methods are called to implement the augmented arithmetic assignments (`+=`, `-=`, `*=`, `@=`, `/=`, `//=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). These methods should attempt to do the operation in-place (modifying *self*) and return the result (which could be, but does not have to be, *self*). If a specific method is not defined, or if that method returns `NotImplemented`, the augmented assignment falls back to the normal methods. For instance, if *x* is an instance of a class with an *__iadd__()* method, `x += y` is equivalent to `x = x.__iadd__(y)` . If *__iadd__()* does not exist, or if `x.__iadd__(y)` returns `NotImplemented`, `x.__add__(y)` and `y.__radd__(x)` are considered, as with the evaluation of `x + y`. In certain situations, augmented assignment can result in unexpected errors (see faq-augmented-assignment-tuple-error), but this behavior is in fact part of the data model.

`object.__neg__`(*self*)

`object.__pos__`(*self*)

object.__abs__(*self*)

object.__invert__(*self*)

呼び出して単項算術演算 (-, +, abs() および ~) を実装します。

object.__complex__(*self*)

object.__int__(*self*)

object.__float__(*self*)

組み込み関数の complex(), int(), float() の実装から呼び出されます。適切な型の値を返さなければなりません。

object.__index__(*self*)

呼び出して operator.index() を実装します。Python が数値オブジェクトを整数オブジェクトに損失なく変換する必要がある場合 (たとえばスライシングや、組み込みの bin() 、hex() 、oct() 関数) は常に呼び出されます。このメソッドがあるとその数値オブジェクトが整数型であることが示唆されます。整数を返さなければなりません。

もし *__int__()*, *__float__()*, *__complex__()* が定義されていない場合、組み込み関数の int(), float(), complex() は *__index__()* にフォールバックします。

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

object.__trunc__(*self*)

object.__floor__(*self*)

object.__ceil__(*self*)

組み込み関数の round() と math モジュール関数の trunc(), floor(), ceil() の実装から呼び出されます。*ndigits* が __round__() に渡されない限りは、これらの全てのメソッドは Integral (たいていは int) に切り詰められたオブジェクトの値を返すべきです。

The built-in function int() falls back to *__trunc__()* if neither *__int__()* nor *__index__()* is defined.

バージョン 3.11 で変更: int() の *__trunc__()* への処理の委譲は非推奨になりました。

3.3.9 with 文とコンテキストマネージャ

コンテキストマネージャ (*context manager*) とは、*with* 文の実行時にランタイムコンテキストを定義するオブジェクトです。コンテキストマネージャは、コードブロックを実行するために必要な入り口および出口の処理を扱います。コンテキストマネージャは通常、with 文（ *with* **文** の章を参照）により起動されますが、これらのメソッドを直接呼び出すことで起動することもできます。

コンテキストマネージャの代表的な使い方としては、様々なグローバル情報の保存および更新、リソースのロックとアンロック、ファイルのオープンとクローズなどが挙げられます。

コンテキストマネージャについてのさらなる情報については、typecontextmanager を参照してください。

object.__enter__(*self*)

コンテキストマネージャのの入り口で実行される処理です。*with* 文は、文の as 節で規定された値を返すこのメソッドを呼び出します。

object.__exit__(*self*, *exc_type*, *exc_value*, *traceback*)

コンテキストマネージャの出口で実行される処理です。パラメータは、コンテキストが終了した原因となった例外について説明しています。コンテキストが例外を送出せず終了した場合は、全ての引き数に None が設定されます。

もし、例外が送出され、かつメソッドが例外を抑制したい場合（すなわち、例外が伝播されるのを防ぎたい場合）、このメソッドは True を返す必要があります。そうでなければ、このメソッドの終了後、例外は通常通り伝播することになります。

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

参考

PEP 343 - "with" **ステートメント**
Python の *with* 文の仕様、背景、および例が記載されています。

3.3.10 クラスパターンマッチの位置引数のカスタマイズ

パターンの中でクラス名を利用する場合、位置引数はデフォルトでは利用できません。MyClass で特別なサポートがないと、case MyClass(x, y) は通常無効です。このようなパターンを利用するには、*__match_args__* 属性をクラスに定義する必要があります。

object.__match_args__

このクラス変数には文字列のタプルがアサイン可能です。このクラスがクラスパターンの位置引数の中で利用されると、それぞれの位置引数は対応する *__match_args__* の中の値をキーワードとする、キーワード引数に変換されます。この属性がない時は、() が設定されているのと同義です。

例えば、もし MyClass.__match_args__ に ("left", "center", "right") が定義されていた場合、case MyClass(x, y) は case MyClass(left=x, center=y) と同義です。パターンの引数の数は、*__match_args__* の要素数と同等かそれ以下でなければならない点に注意してください。もし、多かった場合には、パターンマッチは TypeError を送出します。

Added in version 3.10.

参考

PEP 634 - 構造的パターンマッチ
: match 文の詳細。

3.3.11 Emulating buffer types

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

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

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

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

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

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

Added in version 3.12.

参考

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

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

3.3.12 特殊メソッド検索

カスタムクラスでは、特殊メソッドの暗黙の呼び出しは、オブジェクトのインスタンス辞書ではなく、オブジェクトの型で定義されているときにのみ正しく動作することが保証されます。この動作のため、以下のコードは例外を送出します:

```
>>> 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()
```

この動作の背景となる理由は、*__hash__()* と *__repr__()* といった type オブジェクトを含むすべてのオブジェクトで定義されている特殊メソッドにあります。これらのメソッドの暗黙の検索が通常の検索プロセスを使った場合、type オブジェクト自体に対して実行されたときに失敗してしまいます:

```
>>> 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
```

クラスの非結合メソッドをこのようにして実行しようとすることは、'metaclass confusion' と呼ばれることもあり、特殊メソッドを検索するときはインスタンスをバイパスすることで回避されます:

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

正確性のためにインスタンス属性をスキップするのに加えて、特殊メソッド検索はオブジェクトのメタクラスを含めて、*__getattribute__()* メソッドもバイパスします:

```
>>> class Meta(type):
...     def __getattribute__(*args):
...         print("Metaclass getattribute invoked")
...         return type.__getattribute__(*args)
...
>>> class C(object, metaclass=Meta):
...     def __len__(self):
...         return 10
```

(次のページに続く)

(前のページからの続き)

```
...     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
```

このように *__getattribute__()* 機構をバイパスすることで、特殊メソッドの扱いに関するある程度の自由度と引き換えに (特殊メソッドはインタプリタから一貫して実行されるためにクラスオブジェクトに設定 **しなければならない**)、インタープリタを高速化するための大きな余地が手に入ります。

3.4 コルーチン

3.4.1 待機可能オブジェクト (Awaitable Object)

awaitable オブジェクトは一般的には *__await__()* メソッドが実装されています。*async def* 関数が返す *Coroutine* **オブジェクト** は待機可能です。

注釈

`types.coroutine()` デコレータでデコレータが付けられたジェネレータから返される *generator iterator* オブジェクトも待機可能ですが、*__await__()* は実装されていません。

`object.__await__`(*self*)

iterator を返さなければなりません。このメソッドは *awaitable* オブジェクトを実装するのに使われるべきです。簡単のために、`asyncio.Future` にはこのメソッドが実装され、*await* 式と互換性を持つようになっています。

注釈

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

Added in version 3.5.

参考

待機可能オブジェクトについてより詳しくは **PEP 492** を参照してください。

3.4.2 コルーチンオブジェクト

Coroutine **オブジェクト** は *awaitable* オブジェクトです。*__await__()* を呼び出し、その返り値に対し反復処理をすることでコルーチンの実行を制御できます。コルーチンの実行が完了し制御を戻したとき、イテレータは `StopIteration` を送出し、その例外の `value` 属性に返り値を持たせます。コルーチンが例外を送出した場合は、イテレータにより伝搬されます。コルーチンから `StopIteration` 例外を外に送出すべきではありません。

コルーチンには以下に挙げるメソッドもあり、これらはジェネレータのメソッドからの類似です (**ジェネレータ-イテレータメソッド** を参照してください)。ただし、ジェネレータと違って、コルーチンは反復処理を直接はサポートしていません。

バージョン 3.5.2 で変更: コルーチンで 2 回以上待機 (await) すると `RuntimeError` となります。

`coroutine.send`(*value*)

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

`coroutine.throw`(*value*)

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

コルーチンで指定された例外を送出します。このメソッドは、イテレータにコルーチンを一時停止する *throw()* メソッドがある場合に処理を委任します。そうでない場合には、中断した地点から例外が送出されます。結果 (返り値か `StopIteration` かその他の例外) は、上で解説したような *__await__()* の返り値に対して反復処理を行ったときと同じです。例外がコルーチンの中で捕捉されなかった場合、呼び出し元へ伝搬されます。

バージョン 3.12 で変更: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

`coroutine.close`()

コルーチンが自分自身の後片付けをし終了します。コルーチンが一時停止している場合は、コルーチンを一時停止させたイテレータに *close()* メソッドがあれば、まずはそれに処理を委任します。そして一時停止した地点から `GeneratorExit` が送出され、ただちにコルーチンが自分自身の後片付けを行います。最後に、実行が開始されていなかった場合でも、コルーチンに実行が完了した印を付けます。

コルーチンオブジェクトが破棄されるときには、上記の手順を経て自動的に閉じられます。

3.4.3 非同期イテレータ (Asynchronous Iterator)

非同期イテレータ の `__anext__` メソッドからは非同期のコードが呼べます。

非同期イテレータは *async for* 文の中で使えます。

`object.__aiter__`(*self*)

非同期イテレータ オブジェクトを返さなくてはなりません。

`object.__anext__`(*self*)

イテレータの次の値を返す **待機可能オブジェクト** を返さなければなりません。反復処理が終了したときには `StopAsyncIteration` エラーを送出すべきです。

非同期イテラブルオブジェクトの例:

```
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
```

Added in version 3.5.

バージョン 3.7 で変更: Python 3.7 より前では、*__aiter__()* は **非同期イテレータ** になる *awaitable* を返せました。

Python 3.7 からは、*__aiter__()* は非同期イテレータオブジェクトを返さなければなりません。それ以外のものを返すと `TypeError` になります。

3.4.4 非同期コンテキストマネージャ (Asynchronous Context Manager)

非同期コンテキストマネージャ は、`__aenter__` メソッドと `__aexit__` メソッド内部で実行を一時停止できる **コンテキストマネージャ** です。

非同期コンテキストマネージャは *async with* 文の中で使えます。

`object.__aenter__`(*self*)

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

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

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

非同期コンテキストマネージャクラスの例:

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

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

Added in version 3.5.

脚注

第

FOUR

実行モデル

4.1 プログラムの構造

Python プログラムはコードブロックから構成されます。ブロック (*block*) は、一つのまとまりとして実行される Python プログラムテキストの断片です。モジュール、関数本体、そしてクラス定義はブロックであり、対話的に入力された個々のコマンドもブロックです。スクリプトファイル (インタープリタに標準入力として与えられたり、インタープリタにコマンドライン引数として与えられたファイル) もコードブロックです。スクリプトコマンド (インタープリタのコマンドライン上で `-c` オプションで指定されたコマンド) もコードブロックです。引数 `-m` を使用して、コマンドラインからトップレベルスクリプト (すなわちモジュール `__main__`) として実行されるモジュールもまたコードブロックです。組み込み関数 `eval()` や `exec()` に渡された文字列引数もコードブロックです。

コードブロックは、実行フレーム (*execution frame*) 上で実行されます。実行フレームには、(デバッグに使われる) 管理情報が収められています。また、現在のコードブロックの実行が完了した際に、どのようにプログラムの実行を継続するかを決定しています。

4.2 名前づけと束縛 (naming and binding)

4.2.1 名前の束縛

名前 *(name)* は、オブジェクトを参照します。名前を導入するには、名前への束縛 (name binding) 操作を行います。

以下の構文が名前を束縛します:

- formal parameters to functions,
- クラス定義,
- 関数定義,
- 代入式,

- *targets* that are identifiers if occurring in an assignment:
  - *for* loop header,
  - after `as` in a *with* statement, *except* clause, *except** clause, or in the as-pattern in structural pattern matching,
  - in a capture pattern in structural pattern matching
- *import* 文.
- *type* statements.
- *type parameter lists*.

The `import` statement of the form `from ... import *` binds all names defined in the imported module, except those beginning with an underscore. This form may only be used at the module level.

del 文で指定される対象は、(*del* の意味付けは、実際は名前の解放 (unbind) ですが) 文の目的上、束縛済みのものとみなされます。

代入文や import 文はいずれも、クラスや関数定義、モジュールレベル (トップレベルのコードブロック) 内で起こります。

ある名前がブロック内で束縛されているなら、*nonlocal* や *global* として宣言されていない限り、それはそのブロックのローカル変数 (local variable) です。ある名前がモジュールレベルで束縛されているなら、その名前はグローバル変数 (global variable) です。(モジュールコードブロックの変数は、ローカル変数でも、グローバル変数でもあります。) ある変数があるコードブロック内で使われていて、そのブロックで定義はされていないなら、それは自由変数 (*free variable*) です。

プログラムテキスト中に名前が出現するたびに、その名前が使われている最も内側の関数ブロック中で作成された **束縛** *(binding)* を使って名前の参照が行われます。

4.2.2 名前解決

スコープ (*scope*) は、ブロック内の名前の可視性を決めます。ローカル変数があるブロック内で定義されている場合、変数のスコープはそのブロックを含みます。関数ブロック内で名前の定義を行った場合、その中のブロックが名前に別の束縛を行わない限り、定義ブロック内の全てのブロックを含むようにスコープが拡張されます。

ある名前がコードブロック内で使われると、その名前を最も近傍から囲うようなスコープ (最内スコープ: nearest enclosing scope) を使って束縛の解決を行います。こうしたスコープからなる、あるコードブロック内で参照できるスコープ全ての集合は、ブロックの環境 (*environment*) と呼ばれます。

名前が全く見付からなかったときは、`NameError` 例外が送出されます。現在のスコープが関数のもので、名前が使われる場所でローカル変数がまだ値に束縛されていない場合、`UnboundLocalError` 例外が送出されます。`UnboundLocalError` は `NameError` の子クラスです。

ある名前がコードブロック内のどこかで束縛操作されていたら、そのブロック内で使われるその名前はすべて、現在のブロックへの参照として扱われます。このため、ある名前がそのブロック内で束縛される前に使われるとエラーにつながります。この規則は敏感です。Python には宣言がなく、コードブロックのどこでも名前束縛操作ができます。あるコードブロックにおけるローカル変数は、ブロックのテキスト全体から名前束縛操作を走査することで決定されます。例は UnboundLocalError についての FAQ 項目 を参照してください。

If the *global* statement occurs within a block, all uses of the names specified in the statement refer to the bindings of those names in the top-level namespace. Names are resolved in the top-level namespace by searching the global namespace, i.e. the namespace of the module containing the code block, and the builtins namespace, the namespace of the module `builtins`. The global namespace is searched first. If the names are not found there, the builtins namespace is searched next. If the names are also not found in the builtins namespace, new variables are created in the global namespace. The global statement must precede all uses of the listed names.

global 文は、同じブロックの束縛操作と同じスコープを持ちます。ある自由変数の最内スコープに global 文がある場合、その自由変数はグローバル変数とみなされます。

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

あるモジュールの名前空間は、そのモジュールが最初に import された時に自動的に作成されます。スクリプトの主モジュール (main module) は常に `__main__` と呼ばれます。

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

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

However, the following will succeed:

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

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

4.2.3 Annotation scopes

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

Annotation scopes are used in the following contexts:

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

Annotation scopes differ from function scopes in the following ways:

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

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

バージョン 3.13 で変更: Annotation scopes are also used for type parameter defaults, as introduced by **PEP 696**.

4.2.4 Lazy evaluation

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

例:

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

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

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

```
from typing import Literal

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

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

Added in version 3.12.

4.2.5 組み込みと制限付きの実行

CPython 実装の詳細: ユーザは `__builtins__` に触れるべきではありません; これは厳密に実装の詳細です。組み込みの名前空間の中の値をオーバーライドしたいユーザは、`builtins` モジュールを *import* して、その属性を適切に変更するべきです。

あるコードブロックの実行に関連する組み込み名前空間は、実際にはコードブロックのグローバル名前空間から名前 `__builtins__` を検索することで見つかります; `__builtins__` は辞書かモジュールでなければなりません (後者の場合はモジュールの辞書が使われます)。デフォルトでは、`__main__` モジュール中においては、`__builtins__` は組み込みモジュール `builtins` です; それ以外の任意のモジュールにおいては、`__builtins__` は `builtins` モジュール自身の辞書のエイリアスです。

4.2.6 動的な機能とのやりとり

自由変数の名前解決はコンパイル時でなく実行時に行われます。つまり、以下のコードは 42 を出力します:

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

`eval()` と `exec()` 関数は、名前の解決に、現在の環境の全てを使えるわけではありません。名前は呼び出し側のローカルやグローバル名前空間で解決できます。自由変数は最内名前空間ではなく、グローバル名前空間から解決されます。[*1] `exec()` と `eval()` 関数にはオプションの引数があり、グローバルとローカル名前空間をオーバライドできます。名前空間が一つしか指定されなければ、両方の名前空間として使われます。

4.3 例外

例外とは、コードブロックの通常の制御フローを中断して、エラーやその他の例外的な状況を処理できるようにするための手段です。例外はエラーが検出された時点で **送出** *(raise)* されます; 例外は、エラーが発生部の周辺のコードブロックか、エラーが発生したコードブロック直接または間接的に呼び出しているコードブロックで **処理** *(handle)* することができます。

Python インタプリタは、ランタイムエラー (ゼロ除算など) が検出されると例外を送出します。Python プログラムから、*raise* 文を使って明示的に例外を送出することもできます。例外ハンドラ (exception handler) は、*try* ... *except* 文で指定することができます。*try* 文の *finally* 節を使うとクリーンアップコード (cleanup code) を指定できます。このコードは例外は処理しませんが、先行するコードブロックで例外が起きても起きなくても実行されます。

[*1] この制限は、上記の操作によって実行されるコードが、モジュールをコンパイルしたときには利用できないために起こります。

Python は、エラー処理に ” プログラムの終了 (termination)” モデルを用いています: 例外ハンドラは、プログラムに何が発生したかを把握することができ、ハンドラの外側のレベルに処理を継続することはできますが、(問題のあったコード部分を最初から実行しなおすのでない限り) エラーの原因を修復したり、実行に失敗した操作をやり直すことはできません。

例外が全く処理されないとき、インタプリタはプログラムの実行を終了させるか、対話メインループに処理を戻します。どちらの場合も、例外が `SystemExit` でなければ、スタックのトレースバックを出力します。

例外は、クラスインスタンスによって識別されます。*except* 節はインスタンスのクラスにもとづいて選択されます: これはインスタンスのクラスか、その **非仮想基底クラス** を参照します。このインスタンスはハンドラによって受け取られ、例外条件に関する追加情報を伝えることができます。

注釈

例外のメッセージは、Python API 仕様には含まれていません。メッセージの内容は、ある Python のバージョンと次のバージョンの間で警告なしに変更される可能性があるので、複数バージョンのインタプリタで動作するようなコードは、例外メッセージの内容に依存させるべきではありません。

try 文については、*try* **文** 節、*raise* 文については *raise* **文** 節も参照してください。

脚注

第

FIVE

インポートシステム

ある 1 つの *module* にある Python コードから他のモジュールを **インポート** することで、そこにあるコードへアクセスできるようになります。*import* 文はインポート機構を動かす最も一般的な方法ですが、それが唯一の方法ではありません。`importlib.import_module()` や組み込みの `__import__()` といった関数を使っても、インポート機構を動かすことができます。

import 文は 2 つの処理を連続して行っています; ある名前のモジュールを探し、その検索結果をローカルスコープの名前に束縛します。`import` 文の検索処理は、適切な引数で `__import__()` 関数を呼び出すこととして定義されています。`__import__()` の戻り値は `import` 文の名前束縛処理の実行で使われます。名前束縛処理の厳密な詳細は `import` 文を参照してください。

`__import__()` を直接呼び出すとモジュールの検索のみが行われ、見つかった場合、モジュールの作成処理が行われます。親パッケージのインポートや (`sys.modules` を含む) 様々なキャッシュの更新などの副作用は起きるかもしれませんが、*import* 文のみが名前束縛処理を行います。

import 文が実行されるときには、標準の組み込み関数 `__import__()` が呼ばれます。インポートシステムを呼び出すその他の (`importlib.import_module()` 関数のような) メカニズムは、`__import__()` の呼び出しをバイパスして独自のインポート・セマンティクスを実装している可能性があります。

モジュールが初めてインポートされるとき、Python はそのモジュールを検索し、見付かった場合、モジュールオブジェクトを作成し、初期化します[*1] 。その名前のモジュールが見付からなかった場合、`ModuleNotFoundError` が送出されます。Python には、インポート機構が実行されたときに名前からモジュールを検索する様々な戦略が実装されています。これらの戦略は、これ以降の節で解説される様々なフックを使って、修正したり拡張したりできます。

バージョン 3.3 で変更: インポートシステムが **PEP 302** の第 2 フェーズの完全な実装へ更新されました。もはや暗黙的なインポート機構はありません - インポート機構全体は `sys.meta_path` を通して公開されています。加えて、ネイティブの名前空間パッケージのサポートは実装されています (**PEP 420** を参照) 。

[*1] `types.ModuleType` を参照してください。

5.1 importlib

importlib モジュールはインポート機構とやり取りするための便利な API を提供します。例えば importlib.import_module() は、インポート機構を実行するための組み込みの __import__() よりもシンプルで推奨される API を提供します。より詳細なことは importlib ライブラリのドキュメントを参照してください。

5.2 パッケージ

Python にはモジュールオブジェクトの種類は 1 種類しかなく、Python 、C 、それ以外のもののどれで実装されているかに関係なく、すべてのモジュールはこの種類になります。モジュールの組織化を助け、名前階層を提供するために、Python には **パッケージ** という概念があります。

パッケージはファイルシステムのディレクトリ、モジュールはディレクトリにあるファイルと考えることができますが、パッケージやモジュールはファイルシステムから生まれる必要はないので、この比喩を額面通りに受け取ってはいけません。この文書の目的のために、ディレクトリとファイルという便利な比喩を使うことにします。ファイルシステムのディレクトリのように、パッケージは階層構造を成し、通常のモジュールだけでなく、サブパッケージを含むこともあります。

すべてのパッケージはモジュールですが、すべてのモジュールがパッケージとは限らないことを心に留めておくのが重要です。もしくは他の言い方をすると、パッケージは単なる特別な種類のモジュールであると言えます。特に、__path__ 属性を持つ任意のモジュールはパッケージと見なされます。

すべてのモジュールは名前を持ちます。Python の属性アクセスの文法と同様に、サブパッケージの名前は親パッケージ名とドット記号で区切られます。したがって、email という名前のパッケージや、それが含む email.mime という名前のサブパッケージ、さらにそれに含まれる email.mime.text と言う名前のモジュールを考えることができます。

5.2.1 通常のパッケージ

Python では、**通常のパッケージ** と **名前空間パッケージ** の 2 種類のパッケージが定義されています。通常のパッケージは Python 3.2 以前から存在する伝統的なパッケージです。典型的な通常のパッケージは __init__.py ファイルを含むディレクトリとして実装されます。通常のパッケージがインポートされたとき、この __init__.py ファイルが暗黙的に実行され、それで定義しているオブジェクトがパッケージ名前空間にある名前に束縛されます。__init__.py ファイルは、他のモジュールに書ける Python コードと同じものを含むことができ、モジュールがインポートされたときに Python はモジュールに属性を追加したりします。

例えば、以下のようなファイルシステム配置は、3 つのサブパッケージを持つ最上位の parent パッケージを定義します:

```
parent/
    __init__.py
```

(次のページに続く)

(前のページからの続き)

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

parent.one をインポートすると暗黙的に parent/__init__.py と parent/one/__init__.py が実行されます。その後に parent.two もしくは parent.three をインポートすると、それぞれ parent/two/__init__.py や parent/three/__init__.py が実行されます。

5.2.2 名前空間パッケージ

名前空間パッケージは様々な **ポーション** を寄せ集めたもので、それぞれのポーションはサブパッケージを親パッケージに提供します。ポーションはファイルシステムの別々の場所にあることもあります。ポーションは、zip ファイルの中やネットワーク上や、それ以外のインポート時に Python が探すどこかの場所で見つかることもあります。名前空間パッケージはファイルシステム上のオブジェクトに対応することもあるし、そうでないこともあります; それらは実際の実体のない仮想モジュールです。

名前空間パッケージは、__path__ 属性に普通のリストは使いません。その代わりに独自の iterable 型を使っていて、ポーションの親パッケージのパス (もしくは最上位パッケージのための sys.path) が変わった場合、そのパッケージでの次のインポートの際に、新たに自動でパッケージポーションを検索します。

名前空間パッケージには parent/__init__.py ファイルはありません。それどころか、異なるポーションがそれぞれ提供する複数の parent ディレクトリがインポート検索の際に見つかることもあります。したがって parent/one は物理的に parent/two の隣りにあるとは限りません。その場合、そのパッケージかサブパッケージのうち 1 つがインポートされたとき、Python は最上位の parent パッケージのための名前空間パッケージを作成します。

名前空間パッケージの仕様については **PEP 420** も参照してください。

5.3 検索

検索を始めるためには、Python はインポートされるモジュール (もしくはパッケージですが、ここでの議論の目的においてはささいな違いです) の **完全修飾** 名を必要とします。この名前は、*import* 文の様々な引数や importlib.import_module() および __import__() 関数のパラメータから得られます。

この名前はインポート検索の様々なフェーズで使われ、これは例えば foo.bar.baz のようなドットで区切られたサブモジュールへのパスだったりします。この場合、Python は最初に foo を、次に foo.bar 、そして最後に foo.bar.baz をインポートしようとします。中間のいずれかのインポートに失敗した場合は、ModuleNotFoundError が送出されます。

5.3.1 モジュールキャッシュ

インポート検索で最初に調べる場所は sys.modules です。このマッピングは、中間のパスを含む、これまでにインポートされたすべてのモジュールのキャッシュを提供します。なので foo.bar.baz がインポート済みの場合、sys.modules は foo 、foo.bar 、foo.bar.baz のエントリーを含みます。それぞれのキーはその値として対応するモジュールオブジェクトを持ちます。

インポートではモジュール名は sys.modules から探され、存在した場合は、対応する値がインポートされるべきモジュールであり、この処理は完了します。しかし値が None だった場合、ModuleNotFoundError が送出されます。モジュール名が見付からなかった場合は、Python はモジュールの検索を続けます。

sys.modules は書き込み可能です。キーの削除は対応するモジュールを破壊しない (他のモジュールがそのモジュールへの参照を持っている) かもしれませんが、指定されたモジュールのキャッシュされたエントリーを無効にし、それが次にインポートされたとき Python にそのモジュールを改めて検索させることになります。キーを None に対応付けることもできますが、次にそのモジュールがインポートされるときに ModuleNotFoundError となってしまいます。

たとえモジュールオブジェクトへの参照を保持しておいて、sys.modules にキャッシュされたエントリーを無効にし、その指定したモジュールを再インポートしたとしても、2 つのモジュールオブジェクトは同じでは **ない** ことに注意してください。それとは対照的に、importlib.reload() は **同じ** モジュールオブジェクトを再利用し、モジュールのコードを再実行することで単にモジュールの内容を再初期化するだけです。

5.3.2 ファインダーとローダー

sys.modules に指定されたモジュールが見つからなかった場合は、Python のインポートプロトコルが起動され、モジュールを見つけロードします。このプロトコルは 2 つの概念的なオブジェクト、**ファインダー** と **ローダー** から成ります。ファインダーの仕事は、知っている戦略を使って指定されたモジュールを見つけられるかどうか判断することです。両方のインターフェースを実装しているオブジェクトは **インポーター** と呼ばれます - インポーターは要求されたモジュールがロードできると分かったとき、自分自身を返します。

Python にはデフォルトのファインダーとインポーターがいくつかあります。1 つ目のものは組み込みモジュールの見つけ方を知っていて、2 つ目のものは凍結されたモジュール (訳注: freeze ツールで処理されたモジュールのこと。プログラミング FAQ の「どうしたら Python スクリプトからスタンドアロンバイナリを作れますか？」の項目を参照) の見つけ方を知っています。3 つ目のものは **インポートパス** からモジュールを探します。**インポートパス** はファイルシステムのパスや zip ファイルの位置を示すリストです。このリストは、URL で特定できるもののような、位置を示すことのできる任意のリソースの検索にまで拡張することもできます。

インポート機構は拡張可能なので、モジュール検索の範囲とスコープを拡張するために新しいファインダーを付け加えることができます。

ファインダーは実際にはモジュールをロードしません。指定されたモジュールが見つかった場合、ファインダーは *module spec* (モジュール仕様)、すなわちモジュールのインポート関連の情報をカプセル化したものを返します。モジュールのロード時にインポート機構はそれを利用します。

次の節では、インポート機構を拡張するための新しいファインダーやローダーの作成と登録を含め、ファインダーとローダーのプロトコルについてより詳しく解説します。

バージョン 3.4 で変更: Python の以前のバージョンでは、ファインダーは直接 **ローダー** を返していましたが、現在はローダーを **含む** モジュール仕様を返します。ローダーはインポート中はまだ使われていますが、責任は減りました。

5.3.3 インポートフック

インポート機構は拡張可能なように設計されています; その主となる仕組みは **インポートフック** です。インポートフックには 2 種類あります: **メタフック** と **インポートパスフック** です。

メタフックはインポート処理の最初、`sys.modules` キャッシュの検索以外のインポート処理より前に呼び出されます。これにより、`sys.path` の処理や凍結されたモジュールや組み込みのモジュールでさえも、メタフックで上書きすることができます。メタフックは以下で解説するように、`sys.meta_path` に新しいファインダーオブジェクトを追加することで登録されます。

インポートパスフックは、`sys.path` (もしくは `package.__path__`) の処理の一部として、対応するパス要素を取り扱うところで呼び出されます。インポートパスフックは以下で解説するように、新しい呼び出し可能オブジェクトを `sys.path_hooks` に追加することで登録されます。

5.3.4 メタパス

When the named module is not found in `sys.modules`, Python next searches `sys.meta_path`, which contains a list of meta path finder objects. These finders are queried in order to see if they know how to handle the named module. Meta path finders must implement a method called `find_spec()` which takes three arguments: a name, an import path, and (optionally) a target module. The meta path finder can use any strategy it wants to determine whether it can handle the named module or not.

meta path finder が指定されたモジュールの扱い方を知っている場合は、ファインダは spec オブジェクトを返します。指定されたモジュールを扱えない場合は `None` を返します。`sys.meta_path` に対する処理が spec を返さずにリストの末尾に到達してしまった場合は、`ModuleNotFoundError` を送出します。その他の送出された例外はそのまま呼び出し元に伝播され、インポート処理を異常終了させます。

The `find_spec()` method of meta path finders is called with two or three arguments. The first is the fully qualified name of the module being imported, for example `foo.bar.baz`. The second argument is the path entries to use for the module search. For top-level modules, the second argument is `None`, but for submodules or subpackages, the second argument is the value of the parent package's `__path__` attribute. If the appropriate `__path__` attribute cannot be accessed, a `ModuleNotFoundError` is raised. The third argument is an existing module object that will be the target of loading later. The import system passes in a target module only during reload.

メタパスは、1 回のインポート要求で複数回走査される可能性があります。例えば、関係するモジュールがどれもまだキャッシュされていないとしたときに `foo.bar.baz` をインポートすると、最初は各メタパス・ファインダー (`mpf`) に対して `mpf.find_spec("foo", None, None)` を呼び出して、最上位のインポート処理を行います。`foo` がインポートされた後に、`mpf.find_spec("foo.bar", foo.__path__, None)` を呼び出していく 2 回目のメタパスの走査が行われ、`foo.bar` がインポートされます。`foo.bar` のインポートまで行われたら、最後の走査で `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)` を呼び出していきます。

あるメタパス・ファインダーは最上位のインポートのみサポートしています。これらのインポーターは、2 つ目の引数に `None` 以外のものが渡されたとき、常に `None` を返します。

Python のデフォルトの `sys.meta_path` は 3 つのパスファインダーを持っています。組み込みモジュールのインポートの方法を知っているもの、凍結されたモジュールのインポートの方法を知っているもの、**インポートパス** からのモジュールのインポートの方法を知っているもの (つまり **パスベース・ファインダー**) があります。

バージョン 3.4 で変更: The `find_spec()` method of meta path finders replaced `find_module()`, which is now deprecated. While it will continue to work without change, the import machinery will try it only if the finder does not implement `find_spec()`.

バージョン 3.10 で変更: Use of `find_module()` by the import system now raises `ImportWarning`.

バージョン 3.12 で変更: `find_module()` has been removed. Use `find_spec()` instead.

5.4 ロード

モジュール仕様が見つかった場合、インポート機構はモジュールをロードする時にそれ (およびそれに含まれるローダー) を使います。これは、インポートのロード部分で起こることの近似です:

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

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

(次のページに続く)

(前のページからの続き)

```
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
        raise
return sys.modules[spec.name]
```

以下の詳細に注意してください:

- sys.modules の中に与えられた名前を持つ既存のモジュールオブジェクトがあるなら、import は既にそれを返しているでしょう。
- モジュールは、ローダーがモジュールコードを実行する前に sys.modules に存在しています。モジュールコードが (直接的または間接的に) 自分自身をインポートする可能性があるので、これは重要です; モジュールを sys.modules に追加することで、最悪のケースでは無限の再帰が、そして最良のケースでは複数回のロードが、前もって防止されます。
- ロード処理に失敗した場合、その失敗したモジュールは -- そして、そのモジュールだけが -- sys.modules から取り除かれます。sys.modules キャッシュに既に含まれていたすべてのモジュールと、副作用としてロードに成功したすべてのモジュールは、常にキャッシュに残されます。これはリロードとは対照的で、リロードの場合は失敗したモジュールも sys.modules に残されます。
- **後のセクション** で要約されるように、モジュールが作られてから実行されるまでの間にインポート機構はインポート関連のモジュール属性を設定します (上記擬似コード例の "_init_module_attrs")。
- モジュール実行はモジュールの名前空間が構築されるロードの重要な瞬間です。実行はローダーに完全に委任され、ローダーは何をどのように構築するかを決定することになります。
- ロードの間に作成されて exec_module() に渡されたモジュールは、インポートの終わりに返されるものとは異なるかもしれません[*2]。

バージョン 3.4 で変更: インポートシステムはローダーの定型的な責任を引き継ぎました。これらは以前は importlib.abc.Loader.load_module() メソッドによって実行されました。

[*2] importlib の実装は、戻り値を直接使うことは避けています。その代わりに、モジュール名を調べて sys.modules からモジュールオブジェクトを得ます。こうすることの間接的な効果は、インポートされたモジュールが sys.modules にいる自分自身を置き換えることがあるということです。これは実装依存の動作であり、他の Python 実装では保証されていない動作です。

5.4.1 ローダー

モジュールローダーは、ロードの重要な機能であるモジュール実行機能を提供します。インポート機構は、実行しようとするモジュールオブジェクトを単一の引数として `importlib.abc.Loader.exec_module()` メソッドを呼び出します。`importlib.abc.Loader.exec_module()` から返された任意の値は無視されます。

ローダーは以下の仕様を満たしていなければいけません:

- モジュールが (組み込みモジュールや動的に読み込まれる拡張モジュールではなくて) Python モジュールだった場合、ローダーはモジュールのグローバル名前空間 (`module.__dict__`) で、モジュールのコードを実行すべきです。
- `exec_module()` の呼び出し中に `ImportError` 以外の例外が送出され、伝播されてきたとしても、モジュールをロードできない場合は `ImportError` を送出すべきです。

多くの場合、ファインダーとローダーは同じオブジェクトで構いません; そのような場合では `find_spec()` メソッドは単に `self` (訳注: オブジェクト自身) を返すだけです。

モジュールローダーは、`create_module()` メソッドを実装することでロード中にモジュールオブジェクトを作成することを選択できます。このメソッドは、モジュール仕様を引数に取って、ロード中に使う新しいモジュールオブジェクトを返します。`create_module()` はモジュールオブジェクトに属性を設定する必要はありません。もしこのメソッドが `None` を返すなら、インポート機構は新しいモジュールを自身で作成します。

Added in version 3.4: ローダーの `create_module()` メソッド。

バージョン 3.4 で変更: `load_module()` メソッドは `exec_module()` によって置き換えられ、インポート機構がロードのすべての定型的な処理を引き受けました。

既存のローダーとの互換性のため、もしローダーに `load_module()` メソッドが存在し、かつローダーが `exec_module()` を実装していなければ、インポート機構はローダーの `load_module()` メソッドを使います。しかし、`load_module()` は deprecated であり、ローダーは代わりに `exec_module()` を実装すべきです。

`load_module()` メソッドは、モジュールを実行することに加えて上記で説明されたすべての定型的なロード機能を実施しなければなりません。同じ制約が適用されます。以下は追加の明確化です:

- `sys.modules` に与えられた名前のモジュールが存在している場合、ローダーはその既存のモジュールを使わなければいけません。(そうしないと `importlib.reload()` は正しく動かないでしょう。) 指定されたモジュールが `sys.modules` に存在しない場合、ローダーは新しいモジュールオブジェクトを作成し、`sys.modules` に追加しなければいけません。
- 無限の再帰または複数回のロードを防止するために、ローダーがモジュールコードを実行する前にモジュールは `sys.modules` に存在しなければなりません (*must*)。
- ロード処理に失敗した場合、ローダーは `sys.modules` に追加したモジュールを取り除かなければいけませんが、それはロードに失敗したモジュール **のみ** を、そのモジュールがローダー自身に明示的にロードされた場合に限り、除去しなければなりません。

バージョン 3.5 で変更: exec_module() が定義されていて create_module() が定義されていない場合、DeprecationWarning が送出されるようになりました。

バージョン 3.6 で変更: exec_module() が定義されていて create_module() が定義されていない場合、ImportError が送出されるようになりました。

バージョン 3.10 で変更: load_module() を使用すると ImportWarning が発生します。

5.4.2 サブモジュール

サブモジュールをロードするのにどのようなメカニズム (例えば、importlib API 、import または import-from ステートメント、またはビルトイン関数の __import__) が使われた場合でも、バインディングはサブモジュールオブジェクトを親モジュールの名前空間に配置します。例えば、もしパッケージ spam がサブモジュール foo を持っていた場合、spam.foo をインポートした後は spam は値がサブモジュールに束縛された属性 foo を持ちます。以下のディレクトリ構造を持っているとしましょう:

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

そして spam/__init__.py は以下のようになっているとします:

```
from .foo import Foo
```

このとき、以下を実行することにより spam モジュールの中に foo と Foo に束縛された名前が置かれます:

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

Python の慣れ親しんだ名前束縛ルールからするとこれは驚きかもしれませんが、それは実際インポートシステムの基本的な機能です。不変に保たなければならないのは (上記のインポートの後などで) sys.modules['spam'] と sys.modules['spam.foo'] が存在する場合、後者が前者の foo 属性として存在しなければならないということです。

5.4.3 モジュール仕様

インポート機構は、インポートの間 (特にロードの前) に、個々のモジュールについてのさまざまな情報を扱います。情報のほとんどはすべてのモジュールで共通です。モジュール仕様の目的は、このインポート関連の情報をモジュールの単位でカプセル化することです。

インポートの際にモジュール仕様を使うことは、インポートシステムコンポーネント間、例えばモジュール仕様を作成するファインダーとそれを実行するローダーの間で状態を転送することを可能にします。最も重要なのは、それによってインポート機構がロードの定型的な作業を実行できるようになるということです。これに対して、モジュール仕様なしではローダがその責任を担っていました。

モジュール仕様は、モジュールオブジェクトの `__spec__` 属性として公開されます。モジュール仕様の内容の詳細については `ModuleSpec` を参照してください。

Added in version 3.4.

5.4.4 インポート関連のモジュール属性

インポート機構はロードの間、モジュールの仕様に基づいて、ローダーがモジュールが実行する前に以下の属性を書き込みます。

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

`__name__`

`__name__` 属性はモジュールの完全修飾名に設定されなければなりません。この名前を利用してインポートシステムでモジュールを一意に識別します。

`__loader__`

`__loader__` 属性はモジュールロード時にインポート機構が使用したローダーオブジェクトに設定されなければなりません。この属性は普通は内省用のものですが、ローダー固有の追加機能のために用いることが出来ます。例えばローダー関連のデータの取得です。

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

バージョン 3.12 で変更: The value of `__loader__` is expected to be the same as `__spec__.loader`. The use of `__loader__` is deprecated and slated for removal in Python 3.14.

`__package__`

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

この属性は **PEP 366** で定義されているように、メインモジュールからの明示的な相対インポートを計算するために、`__name__` の代わりに使用されます。

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

バージョン 3.6 で変更: `__package__` の値が `__spec__.parent` と同じ値を持つことを要求されるようになりました。

バージョン 3.10 で変更: `ImportWarning` is raised if import falls back to `__package__` instead of `parent`.

バージョン 3.12 で変更: Raise `DeprecationWarning` instead of `ImportWarning` when falling back to `__package__`.

`__spec__`

`__spec__` 属性はモジュールロード時に使用されたモジュールスペックに設定されなければなりません。`__spec__` を適切に設定すると **インタープリタ起動中に初期化されるモジュール** にも同様に適用されます。例外は `__main__` で、`__spec__` は **場合によっては** *None* **に設定されます**。

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

Added in version 3.4.

バージョン 3.6 で変更: `__package__` が定義されていないときに `__spec__.parent` がフォールバックとして使われるようになりました。

`__path__`

モジュールが (通常のまたは名前空間) パッケージの場合、モジュールオブジェクトの `__path__` 属性が設定される必要があります。値はイテレート可能でなければなりませんが、`__path__` に意味がない場合は空でも構いません。`__path__` が空でない場合、イテレート時に文字列を生成しなければなりません。`__path__` のセマンティクスの詳細は **下記** の通りです。

パッケージでないモジュールは `__path__` 属性を持ってはいけません。

`__file__`

`__cached__`

`__file__` is optional (if set, value must be a string). It indicates the pathname of the file from which the module was loaded (if loaded from a file), or the pathname of the shared library file for extension modules loaded dynamically from a shared library. It might be missing for certain types of modules, such as C modules that are statically linked into the interpreter, and the import system may opt to leave it unset if it has no semantic meaning (e.g. a module loaded from a database).

もし `__file__` を設定するなら、`__cached__` 属性もコードのコンパイルされたバージョンのどれか (例えば、バイトコンパイルされたファイル) へのパスに設定できます。この属性を設定するにあたってファイル

が存在する必要はありません; パスは、単にコンパイルされたファイルが存在するかもしれない場所を示しているだけです (**PEP 3147** を参照)。

`__file__` が設定されていないときにも `__cached__` は設定できることに注意してください。ただし、そのシナリオはかなり変則的です。究極的には、ローダーとは (`__file__` と `__cached__` の由来である) ファインダーが提供するモジュール仕様を利用するものです。したがって、もしローダーがキャッシュされたモジュールからロードする一方でファイルからはロードしないなら、その変則的なシナリオは適切でしょう。

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

5.4.5 module.__path__

定義より、モジュールに `__path__` 属性があれば、そのモジュールはパッケージとなります。

パッケージの `__path__` 属性は、そのサブパッケージのインポート中に使われます。インポート機構の内部では、それは `sys.path` とほとんど同じように機能します。つまり、インポート中にモジュールを探す場所のリストを提供します。しかし、一般的に `__path__` は `sys.path` よりも制約が強いです。

`__path__` は文字列の iterable でなければいけませんが、空でも構いません。`sys.path` と同じ規則がパッケージの `__path__` にも適用され、パッケージの `__path__` を走査するときに (後で解説する) `sys.path_hooks` が考慮に入れられます。

パッケージの `__init__.py` ファイルは、パッケージの `__path__` 属性を設定もしくは変更することがあり、これが **PEP 420** 以前の名前空間パッケージの典型的な実装方法でした。**PEP 420** の採択により、もはや名前空間パッケージは、`__path__` を操作するコードだけを含む `__init__.py` ファイルを提供する必要がなくなりました; インポート機構は、名前空間パッケージに対し自動的に適切な `__path__` をセットします。

5.4.6 モジュールの repr

デフォルトでは、すべてのモジュールは利用可能な repr を持っています。ただしこれは、これまでに説明した属性の設定内容に依存しており、モジュール仕様によってモジュールオブジェクトの repr をより明示的に制御することができます。

もしモジュールが仕様 (`__spec__`) を持っていれば、インポート機構はそこから repr を生成しようとします。もしそれが失敗するか、または仕様が存在しなければ、インポートシステムはモジュールで入手可能なあらゆる情報を使ってデフォルトの repr を構築します。それは `module.__name__`, `module.__file__`, `module.__loader__` を (足りない情報についてはデフォルト値を使って補いながら) repr への入力として使おうと試みます。

これが使われている正確な規則です:

- モジュールが `__spec__` 属性を持っていれば、仕様に含まれる情報が repr を生成するために使われます。”name”, ”loader”, ”origin”, ”has_location” 属性が参照されます。
- モジュールに `__file__` 属性がある場合は、モジュールの repr の一部として使われます。

- モジュールに `__file__` はないが `__loader__` があり、その値が `None` ではない場合は、ローダーの repr がモジュールの repr の一部として使われます。
- そうでなければ、単にモジュールの `__name__` を repr の中で使います。

バージョン 3.12 で変更: Use of `module_repr()`, having been deprecated since Python 3.4, was removed in Python 3.12 and is no longer called during the resolution of a module's repr.

5.4.7 キャッシュされたバイトコードの無効化

Before Python loads cached bytecode from a `.pyc` file, it checks whether the cache is up-to-date with the source `.py` file. By default, Python does this by storing the source's last-modified timestamp and size in the cache file when writing it. At runtime, the import system then validates the cache file by checking the stored metadata in the cache file against the source's metadata.

Python also supports "hash-based" cache files, which store a hash of the source file's contents rather than its metadata. There are two variants of hash-based `.pyc` files: checked and unchecked. For checked hash-based `.pyc` files, Python validates the cache file by hashing the source file and comparing the resulting hash with the hash in the cache file. If a checked hash-based cache file is found to be invalid, Python regenerates it and writes a new checked hash-based cache file. For unchecked hash-based `.pyc` files, Python simply assumes the cache file is valid if it exists. Hash-based `.pyc` files validation behavior may be overridden with the `--check-hash-based-pycs` flag.

バージョン 3.7 で変更: Added hash-based `.pyc` files. Previously, Python only supported timestamp-based invalidation of bytecode caches.

5.5 パスベース・ファインダー

上で触れた通り、Python にはいくつかのデフォルトのメタパス・ファインダーが備わっています。そのうちの 1 つは **パスベース・ファインダー** (`PathFinder`) と呼ばれ、**パスエントリ** のリストである **インポートパス** を検索します。それぞれのパスエントリは、モジュールを探す場所を指しています。

パスベース・ファインダー自体は何かのインポート方法を知っているわけではありません。その代わりに、個々のパスエントリを走査し、それぞれに特定の種類のパスの扱いを知っているパスエントリ・ファインダーを関連付けます。

デフォルトのパスエントリ・ファインダーは、ファイルシステム上のモジュールを見つけるためのすべてのセマンティクスを実装しています。それは Python ソースコード (`.py` ファイル) 、Python バイトコード (`.pyc` ファイル) 、共有ライブラリ (例えば `.so` ファイル) などの特別なファイルタイプを処理します。標準ライブラリの `zipimport` モジュールによってサポートされる場合は、デフォルトのパスエントリ・ファインダーは (共有ライブラリ以外の) すべてのファイルタイプの zip ファイルからのロードも扱います。

パスエントリはファイルシステム上の場所に限定される必要はありません。URL やデータベースクエリやその他文字列で指定できる場所を参照することも可能です。

パスベース・ファインダーにはフックやプロトコルを追加することができ、それによって検索可能なパスエントリの種類を拡張し、カスタマイズすることができます。例えば、ネットワーク上の URL をパスエントリとしてサポートしたい場合、web 上のモジュールを見つけるために HTTP の取り扱い方を実装したフックを書くことができます。この (呼び出し可能オブジェクトである) フックは、下で解説するプロトコルをサポートする **パスエントリ・ファインダー** を返します。このプロトコルは web からモジュールのローダーを取得するのに使われます。

警告の言葉: この節と前の節の両方で **ファインダー** という言葉が、**メタパス・ファインダー** と **パスエントリ・ファインダー** という用語で区別されて使われています。これら 2 種類のファインダーは非常に似ており、似たプロトコルをサポートし、インポート処理で同じように機能しますが、微妙に異なっているのを心に留めておくのは重要です。特に、メタパス・ファインダーはインポート処理の開始時、`sys.meta_path` の走査が動くときに動作します。

それとは対照的に、パスエントリ・ファインダーはある意味でパスベース・ファインダーの実装詳細であり、実際 `sys.meta_path` からパスベース・ファインダーが取り除かれた場合、パスエントリ・ファインダーの実装は何も実行されないでしょう。

5.5.1 パスエントリ・ファインダー

パスベース・ファインダー には、文字列 **パスエントリ** で指定された場所の Python モジュールや Python パッケージを見つけ、ロードする責任があります。ほとんどのパスエントリはファイルシステム上の場所を指定していますが、そこに制限される必要はありません。

メタパス・ファインダーとして、**パスベース・ファインダー** には前に解説した `find_spec()` プロトコルが実装されていますが、これに加えて **インポートパス** からモジュールを見つけ、ロードする方法をカスタマイズするために使えるフックを提供しています。

パスベース・ファインダー は `sys.path` 、`sys.path_hooks` 、`sys.path_importer_cache` という 3 つの変数を使います。さらにパッケージオブジェクトの `__path__` 属性も使います。これらによって、インポート処理をカスタマイズする方法が提供されます。

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

パスベース・ファインダー は **メタパス・ファインダー** なので、インポート機構は、前で解説したパスベース・ファインダーの `find_spec()` メソッドを呼び出すことで **インポートパス** の検索を始めます。`path` 引数が `find_spec()` に渡されたときは、それは走査するパス文字列のリスト - 典型的にはそのパッケージの中でインポートしているパッケージの `__path__` 属性になります。`path` 引数が `None` だった場合、それは最上位のイン

ポートであることを示していて、sys.path が使われます。

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

path entry がキャッシュの中に無かった場合、path based finder は sys.path_hooks の中の呼び出し可能オブジェクトを全て辿ります。このリストのそれぞれの *path entry* **フック** は、検索する path entry という引数 1 つを渡して呼び出されます。その呼び出し可能オブジェクトは path entry を扱える *path entry finder* を返すか、ImportError を送出します。ImportError は、フックが *path entry* のための *path entry finder* を探せないことを報せるために path based finder が使います。この例外は処理されず、*import path* を辿っていく処理が続けられます。フックは引数として文字列またはバイト列オブジェクトを期待します; バイト列オブジェクトのエンコーディングはフックに任されていて (例えば、ファイルシステムのエンコーディングの UTF-8 やそれ以外などです) 、フックが引数をデコードできなかった場合は ImportError を送出すべきです。

sys.path_hooks を辿る処理が **パスエントリ・ファインダー** を何も返さずに終わった場合、パスベース・ファインダーの find_spec() メソッドは、sys.path_importer_cache に (このパスエントリに対するファインダーが存在しないことを示すために) None を保存し、**メタパス・ファインダー** はモジュールが見つからなかったことを伝えるために None を返します。

sys.path_hooks 上の **パスエントリフック** 呼び出し可能オブジェクトの戻り値のいずれかが **パスエントリ・ファインダー であった** 場合、後で出てくるモジュール仕様を探すためのプロトコルが使われ、それがモジュールをロードするために使われます。

(空の文字列によって表される) 現在のディレクトリは、sys.path の他のエントリとは多少異なる方法で処理されます。まず、現在のディレクトリが存在しないことが判明した場合、sys.path_importer_cache には何も追加されません。次に、現在のディレクトリに対する値は個々のモジュールのルックアップで毎回新たに検索されます。3 番目に、sys.path_importer_cache に使われ、importlib.machinery.PathFinder.find_spec() が返すパスは、実際のディレクトリであって空の文字列ではありません。

5.5.2 パスエントリ・ファインダー・プロトコル

モジュールと初期化されたパッケージのインポートをサポートするため、および名前空間パッケージのポーションとして提供するために、パスエントリ・ファインダーは find_spec() メソッドを実装しなければいけません。

find_spec() は 2 つの引数を取ります。インポートしようとしているモジュールの完全修飾名と、(オプションの) 対象モジュールです。*find_spec()* はモジュールに対応する完全に初期化 (populated) された仕様を返します。この仕様は (1 つの例外を除いて) 常に "loader" セットを持っています。

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

バージョン 3.4 で変更: `find_spec()` replaced `find_loader()` and `find_module()`, both of which are now deprecated, but will be used if `find_spec()` is not defined.

古いパスエントリ・ファインダーの中には、`find_spec()` の代わりにこれら 2 つの deperecated なメソッドのうちのいずれかを実装しているものがあるかもしれません。これらのメソッドは後方互換性のためにまだ考慮されています。しかし、パスエントリ・ファインダーに `find_spec()` が実装されていれば、古いメソッドは無視されます。

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

他のインポート機構の実装に対する後方互換性のために、多くのパスエントリ・ファインダーは、メタパス・ファインダーがサポートするのと同じ伝統的な `find_module()` メソッドもサポートしています。しかし、パスエントリ・ファインダーの `find_module()` メソッドは、決して `path` 引数では呼び出されません (このメソッドは、パスフックの最初の呼び出しから適切なパス情報を記録する動作が期待されています)。

パスエントリ・ファインダーの `find_module()` メソッドは deprecated です。なぜなら、その方法ではパスエントリ・ファインダーが名前空間パッケージに対してポーションを提供することができないからです。もし `find_loader()` と `find_module()` の両方がパスエントリ・ファインダーに存在したら、インポートシステムは常に `find_module()` よりも `find_loader()` を優先して呼び出します。

バージョン 3.10 で変更: Calls to `find_module()` and `find_loader()` by the import system will raise `ImportWarning`.

バージョン 3.12 で変更: `find_module()` and `find_loader()` have been removed.

5.6 標準のインポートシステムを置き換える

インポートシステム全体を置き換えるための最も信頼性のある仕組みは、`sys.meta_path` のデフォルトの内容を削除し、全部をカスタムのメタパスフックで置き換えるものです。

もし、import 文の動作だけを変更し、インポートシステムにアクセスする他の API には影響を与えなくてもよければ、組み込みの `__import__()` 関数を置き換えるだけで十分です。この手法は、ある 1 つのモジュール内だけで import 文の動作を変更するのにも用いられます。

(標準のインポートシステム全体を停止するのではなく) すでにメタパスにいるフックからあるモジュールのインポートを選択的に防ぐためには、`find_spec()` から `None` を返す代わりに、直接 `ModuleNotFoundError` を送出するだけで十分です。`None` を返すのはメタパスの走査を続けるべきであることを意味しますが、例外を送出するとすぐに走査を打ち切ります。

5.7 Package Relative Imports

Relative imports use leading dots. A single leading dot indicates a relative import, starting with the current package. Two or more leading dots indicate a relative import to the parent(s) of the current package, one level per dot after the first. For example, given the following package layout:

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

In either `subpackage1/moduleX.py` or `subpackage1/__init__.py`, the following are valid relative imports:

```
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
```

Absolute imports may use either the `import <>` or `from <> import <>` syntax, but relative imports may only use the second form; the reason for this is that:

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

should expose `XXX.YYY.ZZZ` as a usable expression, but .moduleY is not a valid expression.

5.8 ___main___ に対する特別な考慮

`__main__` モジュールは、Python のインポートシステムに関連する特別なケースです。**他の場所** で言及されているように、`__main__` モジュールは `sys` や `builtins` などと同様にインタプリタースタートアップで直接初期化されます。しかし、前者 2 つのモジュールと違って、`__main__` は厳密にはビルトインのモジュールとしての資格を持っていません。これは、`__main__` が初期化される方法がインタプリタが起動されるときのフラグやその他のオプションに依存するためです。

5.8.1 __main__.__spec__

__main__ がどのように初期化されるかに依存して、__main__.__spec__ は適切に設定されることもあれば None になることもあります。

Python が -m オプションを付けて実行された場合には、__spec__ は対応するモジュールまたはパッケージのモジュール仕様に設定されます。また、ディレクトリや zip ファイル、または他の sys.path エントリを実行する処理の一部として __main__ モジュールがロードされる場合にも __spec__ が生成 (populate) されます。

それ以外のケース では、__main__.__spec__ は None に設定されます。これは、__main__ を生成 (populate) するために使われたコードがインポート可能なモジュールと直接一致していないためです:

- 対話プロンプト
- -c オプション
- stdin から起動された場合
- ソースファイルやバイトコードファイルから直接起動された場合

最後のケースでは、たとえ技術的にはファイルがモジュールとして直接インポートできた **としても** __main__.__spec__ は常に None になることに注意してください。もし __main__ において有効なモジュールメタデータが必要なら -m スイッチを使ってください。

__main__ がインポート可能なモジュールと一致し、__main__.__spec__ がそれに応じて設定されていたとしても、それでもなお、この 2 つのモジュールは別物とみなされることに注意してください。これは、if __name__ == "__main__": チェックによって保証されるブロックは、__main__ 名前空間を生成 (populate) するためにモジュールが使用される時にだけ実行され、通常のインポート時には実行されない、という事実に起因しています。

5.9 参考資料

Python の初期の頃からすると、インポート機構は目覚ましい発展を遂げました。一部細かいところがドキュメントが書かれたときから変わってはいますが、最初期の *パッケージの仕様* はまだ読むことができます。

オリジナルの sys.meta_path の仕様は **PEP 302** で、その後継となる拡張が **PEP 420** です。

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

PEP 366 は、メインモジュールでの明示的な相対インポートのために追加した __package__ 属性の解説をしています。

PEP 328 は絶対インポート、明示的な相対インポート、および、当初 __name__ で提案し、後に **PEP 366** が __package__ で定めた仕様を導入しました。

PEP 338 はモジュールをスクリプトとして実行するときの仕様を定めています。

PEP 451 は、モジュール仕様オブジェクトにおけるモジュール毎のインポート状態のカプセル化を追加しています。また、ローダーの定型的な責任のほとんどをインポート機構に肩代わりさせています。これらの変更により、インポートシステムのいくつかの API が deprecate され、またファインダーとローダーには新しいメソッドが追加されました。

脚注

第

SIX

式 (EXPRESSION)

この章では、Python の式における個々の要素の意味について解説します。

表記法に関する注意: この章と以降の章での拡張 BNF (extended BNF) 表記は、字句解析規則ではなく、構文規則を記述するために用いられています。ある構文規則 (のある表現方法) が、以下の形式

```
name   ::=    othername
```

で記述されていて、この構文特有の意味付け (semantics) が記述されていない場合、`name` の形式をとる構文の意味付けは `othername` の意味付けと同じになります。

6.1 算術変換 (arithmetic conversion)

以下の算術演算子の記述で、「数値引数は共通の型に変換されます」と書かれているとき、組み込み型に対する演算子の実装は以下の通りに動作します:

- 片方の引数が複素数型であれば、他方は複素数型に変換されます;
- otherwise, if either argument is a floating-point number, the other is converted to floating point;
- それ以外場合は、両方の引数は整数でなければならず、変換の必要はありません。

特定の演算子 ('%' 演算子の左引数としての文字列) には、さらに別の規則が適用されます。拡張は、それ自身の型変換のふるまいを定義していなければなりません。

6.2 アトム、原子的要素 (atom)

atom は、式の一番基本的な要素です。もっとも単純な atom は、識別子またはリテラルです。丸括弧、角括弧、または波括弧で囲われた形式 (form) もまた、構文上アトムに分類されます。atom の構文は以下のようになります:

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

6.2.1 識別子 (identifier、または名前 (name))

アトムの形になっている識別子 (identifier) は名前 (name) です。字句定義については **識別子** *(identifier)* **およびキーワード** *(keyword)* 節を、名前付けや束縛については **名前づけと束縛** *(naming and binding)* 節を参照してください。

名前があるオブジェクトに束縛されている場合、名前 atom を評価するとそのオブジェクトになります。名前が束縛されていない場合、atom を評価しようとすると `NameError` 例外を送出します。

Private name mangling

When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a *private name* of that class.

参考

The *class specifications*.

More precisely, private names are transformed to a longer form before code is generated for them. If the transformed name is longer than 255 characters, implementation-defined truncation may happen.

The transformation is independent of the syntactical context in which the identifier is used but only the following private identifiers are mangled:

- Any name used as the name of a variable that is assigned or read or any name of an attribute being accessed.

  The `__name__` attribute of nested functions, classes, and type aliases is however not mangled.

- The name of imported modules, e.g., `__spam` in `import __spam`. If the module is part of a package (i.e., its name contains a dot), the name is *not* mangled, e.g., the `__foo` in `import __foo.bar` is not

mangled.

- The name of an imported member, e.g., `__f` in `from spam import __f`.

The transformation rule is defined as follows:

- The class name, with leading underscores removed and a single leading underscore inserted, is inserted in front of the identifier, e.g., the identifier `__spam` occurring in a class named `Foo`, `_Foo` or `__Foo` is transformed to `_Foo__spam`.
- If the class name consists only of underscores, the transformation is the identity, e.g., the identifier `__spam` occurring in a class named `_` or `__` is left as is.

6.2.2 リテラル

Python では、文字列やバイト列リテラルと、様々な数値リテラルをサポートしています:

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

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

リテラルは全て変更不能なデータ型に対応します。このため、オブジェクトのアイデンティティはオブジェクトの値ほど重要ではありません。同じ値を持つ複数のリテラルを評価した場合、(それらのリテラルがプログラムの同じ場所由来のものであっても、そうでなくても) 同じオブジェクトを指しているか、まったく同じ値を持つ別のオブジェクトになります。

6.2.3 丸括弧形式 (parenthesized form)

丸括弧形式とは、式リストの一形態で、丸括弧で囲ったものです:

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

丸括弧で囲われた式のリストは、個々の式が表現するものになります: リスト内に少なくとも一つのカンマが入っていた場合、タプルになります; そうでない場合、式のリストを構成している単一の式自体の値になります。

中身が空の丸括弧のペアは、空のタプルオブジェクトを表します。タプルは変更不能なので、リテラルと同じ規則が適用されます (すなわち、空のタプルが二箇所で使われると、それらは同じオブジェクトになることもあるし、

ならないこともあります)。

タプルは丸括弧で作成されるのではなく、カンマによって作成されることに注意してください。例外は空のタプルで、この場合には丸括弧が **必要です** --- 丸括弧のつかない "何も記述しない式 (nothing)" を使えるようにしてしまうと、文法があいまいなものになってしまい、よくあるタイプミスが検出されなくなってしまいます。

6.2.4 リスト、集合、辞書の表示

リスト、集合、辞書を構築するために、Python は "表示 (display)" と呼ばれる特別な構文を提供していて、次の二種類ずつがあります:

- コンテナの内容を明示的に列挙する
- **内包表記** *(comprehension)* と呼ばれる、ループ処理とフィルター処理の組み合わせを用いた計算結果

内包表記の共通の構文要素は次の通りです:

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

内包表記はまず単一の式、続いて少なくとも 1 個の `for` 節、さらに続いて 0 個以上の `for` 節あるいは `if` 節からなります。この場合、各々の `for` 節や `if` 節を、左から右へ深くなっていくネストしたブロックとみなし、ネストの最内のブロックに到達するごとに内包表記の先頭にある式を評価した結果が、最終的にできあがるコンテナの各要素になります。

ただし、最も左にある `for` 節のイテラブル式を除いて、内包表記は暗黙的にネストされた個別のスコープで実行されます。この仕組みのおかげで、対象のリスト内で代入された名前が外側のスコープに "漏れる" ことはありません。

最も左にある `for` 節のイテラブル式は、それを直接囲んでいるスコープでそのまま評価され、暗黙的な入れ子のスコープに引数として渡されます。後に続く `for` 節と、最も左にある `for` 節のフィルター条件はイテラブル式を直接囲んでいるスコープでは評価できません。というのは、それらは最も左のイテラブルから得られる値に依存しているかもしれないからです。例えば次の通りです: `[x*y for x in range(10) for y in range(x, x+10)]` 。

内包表記が常に適切な型のコンテナになるのを保証するために、`yield` 式や `yield from` 式は暗黙的な入れ子のスコープでは禁止されています。

Since Python 3.6, in an *`async def`* function, an `async for` clause may be used to iterate over a *asynchronous iterator*. A comprehension in an `async def` function may consist of either a `for` or `async for` clause following the leading expression, may contain additional `for` or `async for` clauses, and may also use *`await`* expressions.

If a comprehension contains `async for` clauses, or if it contains `await` expressions or other asynchronous comprehensions anywhere except the iterable expression in the leftmost `for` clause, it is called an *asynchronous comprehension*. An asynchronous comprehension may suspend the execution of the coroutine function in which it appears. See also **PEP 530**.

Added in version 3.6: 非同期内包表記が導入されました。

バージョン 3.8 で変更: `yield` および `yield from` は暗黙的な入れ子のスコープでは禁止となりました。

バージョン 3.11 で変更: Asynchronous comprehensions are now allowed inside comprehensions in asynchronous functions. Outer comprehensions implicitly become asynchronous.

6.2.5 リスト表示

リスト表示は、角括弧で囲われた式の系列です。系列は空の系列であってもかまいません:

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

リスト表示は、新しいリストオブジェクトを与えます。リストの内容は、式のリストまたはリスト内包表記 (list comprehension) で指定されます。カンマで区切られた式のリストが与えられたときは、それらの各要素は左から右へと順に評価され、その順にリスト内に配置されます。内包表記が与えられたときは、内包表記の結果の要素でリストが構成されます。

6.2.6 集合表示

集合表示は波括弧で表され、キーと値を分けるコロンがないことで辞書表現と区別されます:

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

集合表示は、新しいミュータブルな集合オブジェクトを与えます。集合の内容は、式の並びまたは内包表記によって指定されます。カンマ区切りの式のリストが与えられたときは、その要素は左から右へ順に評価され、集合オブジェクトに加えられます。内包表記が与えられたときは、内包表記の結果の要素で集合が構成されます。

空集合は `{}` で構成できません。このリテラルは空の辞書を構成します。

6.2.7 辞書表示

A dictionary display is a possibly empty series of dict items (key/value pairs) enclosed in curly braces:

```
dict_display        ::=   "{" [dict_item_list | dict_comprehension] "}"
dict_item_list      ::=   dict_item ("," dict_item)* [","]
dict_item           ::=   expression ":" expression | "**" or_expr
dict_comprehension  ::=   expression ":" expression comp_for
```

辞書表示は、新たな辞書オブジェクトを表します。

If a comma-separated sequence of dict items is given, they are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding value. This means that you can specify the same key multiple times in the dict item list, and the final dictionary's value for that key will be the last one given.

A double asterisk `**` denotes *dictionary unpacking*. Its operand must be a *mapping*. Each mapping item is added to the new dictionary. Later values replace values already set by earlier dict items and earlier dictionary unpackings.

Added in version 3.5: 辞書表示のアンパックは最初に **PEP 448** で提案されました。

辞書内包表記は、リストや集合の内包表記とは対照的に、通常の "for" や "if" 節の前に、コロンで分けられた 2 つの式が必要です。内包表記が起動すると、結果のキーと値の要素が、作られた順に新しい辞書に挿入されます。

Restrictions on the types of the key values are listed earlier in section **標準型の階層**. (To summarize, the key type should be *hashable*, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last value (textually rightmost in the display) stored for a given key value prevails.

バージョン 3.8 で変更: Python 3.8 より前のバージョンでは、辞書内包表記において、キーと値の評価順序は明示されていませんでした。CPython では、値がキーより先に評価されていました。バージョン 3.8 からは **PEP 572** で提案されているように、キーが値より先に評価されます。

6.2.8 ジェネレータ式

ジェネレータ式 (generator expression) とは、丸括弧を使ったコンパクトなジェネレータ表記法です:

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

ジェネレータ式は新たなジェネレータオブジェクトを与えます。この構文は内包表記とほぼ同じですが、角括弧や波括弧ではなく、丸括弧で囲まれます。

ジェネレータ式の中で使われている変数は、(通常のジェネレータと同じように) そのジェネレータオブジェクトに対して *__next__()* メソッドが呼ばれるときまで評価が遅延されます。ただし、最も左にある `for` 節のイテラブル式は直ちに評価されます。そのためそこで生じたエラーは、最初の値が得られた時点ではなく、ジェネレータ式が定義された時点で発せられます。後に続く `for` 節と、最も左にある `for` 節のフィルター条件はイテラブル式を直接囲んでいるスコープでは評価できません。というのは、それらは最も左のイテラブルから得られる値に依存しているかもしれないからです。例えば次の通りです: `(x*y for x in range(10) for y in range(x, x+10))`。

関数の唯一の引数として渡す場合には、丸括弧を省略できます。詳しくは **呼び出し** *(call)* 節を参照してください。

ジェネレータ式自身の期待される動作を妨げないために、`yield` 式や `yield from` 式は暗黙的に定義されたジェネレータでは禁止されています。

ジェネレータ式が `async for` 節あるいは *await* 式を含んでいる場合、それは **非同期ジェネレータ式** と呼ばれます。非同期ジェネレータ式は、非同期イテレータである新しい非同期ジェネレータオブジェクトを返します (**非同期イテレータ** *(Asynchronous Iterator)* を参照してください)。

Added in version 3.6: 非同期ジェネレータ式が導入されました。

バージョン 3.7 で変更: Python 3.7 より前では、非同期ジェネレータ式は *async def* コルーチンでしか使えませんでした。3.7 からは、任意の関数で非同期ジェネレータ式が使えるようになりました。

バージョン 3.8 で変更: `yield` および `yield from` は暗黙的な入れ子のスコープでは禁止となりました。

6.2.9 Yield 式

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

yield 式は **ジェネレータ** 関数や **非同期ジェネレータ** 関数を定義するときに使われます。従って、関数定義の本体でのみ使えます。関数の本体で yield 式 を使用するとその関数はジェネレータ関数になり、*async def* 関数の本体で使用するとそのコルーチン関数は非同期ジェネレータ関数になります。例えば次のようになります:

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

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

含まれているスコープの副作用のため、`yield` 式は暗黙的に定義されたスコープの一部として内包表記やジェネレータ式を実装するのに使うことは許可されていません。

バージョン 3.8 で変更: yield 式は、暗黙的な入れ子のスコープで内包表記やジェネレータ式を実装するための使用が禁止になりました。

ジェネレータ関数についてはすぐ下で説明されています。非同期ジェネレータ関数は、**非同期ジェネレータ関数** *(asynchronous generator function)* 節に分けて説明されています。

ジェネレータ関数が呼び出された時、ジェネレータとしてのイテレータを返します。ジェネレータはその後ジェネレータ関数の実行を制御します。ジェネレータのメソッドが呼び出されると実行が開始されます。開始されると、最初の yield 式まで処理して一時停止し、呼び出し元へ *expression_list* の値を、または *expression_list* が省略されていれば None を返します。ここで言う一時停止とは、ローカル変数の束縛、命令ポインタや内部の評価スタック、そして例外処理のを含むすべてのローカル状態が保持されることを意味します。再度、ジェネレータのメソッドが呼び出されて実行を再開した時、ジェネレータは yield 式がただの外部呼び出しであったかのように処理を継続します。再開後の yield 式の値は実行を再開するメソッドに依存します。*__next__()* を使用した場合 (一般に *for* 文や組み込み関数 next() など) の結果は None となり、*send()* を使用した場合はそのメソッドに渡された値が結果になります。

これまで説明した内容から、ジェネレータ関数はコルーチンにとてもよく似ています。ジェネレータ関数は何度も生成し、1 つ以上のエントリポイントを持ち、その実行は一時停止されます。ジェネレータ関数は yield した後で実行の継続を制御できないことが唯一の違いです。その制御は常にジェネレータの呼び出し元へ移されます。

yield 式は *try* 構造内で使用できます。ジェネレータの (参照カウントがゼロに達するか、ガベージコレクションによる) 完了前に再開されない場合、ジェネレータ-イテレータの *close()* メソッドが呼ばれ、*finally* 節が実行されます。

yield from <expr> を使用した場合、与えられた式はイテラブルでなければなりません。そのイテラブルをイテレートすることで生成された値は現在のジェネレータのメソッドの呼び出し元へ直接渡されます。*send()* で渡されたあらゆる値と *throw()* で渡されたあらゆる例外は根底のイテレータに適切なメソッドがあれば渡されます。適切なメソッドがない場合、*send()* は AttributeError か TypeError を、*throw()* は渡された例外を即座に送出します。

根底のイテレータの完了時、引き起こされた StopIteration インスタンスの value 属性はその yield 式の値となります。StopIteration を起こす際に明示的にセットされるか、サブイテレータがジェネレータであれば (サブイテレータからかえる値で) 自動的にセットされるかのどちらかです。

バージョン 3.3 で変更: サブイテレータに制御フローを委譲するために yield from <expr> が追加されました。

yield 式が代入文の単独の右辺式であるとき、括弧は省略できます。

参考

PEP 255 - 単純なジェネレータ
Python へのジェネレータと *yield* 文の導入提案。

PEP 342 - 拡張されたジェネレータを用いたコルーチン シ

ンプルなコルーチンとして利用できるように、ジェネレータの構文と API を拡張する提案。

PEP 380 - **サブジェネレータへの委譲構文** サ

ブジェネレータの委譲を簡単にするための、*yield_from* 構文の導入提案。

PEP 525 - **非同期ジェネレータ** コ

ルーチン関数へのジェネレータの実装能力の追加による **PEP 492** の拡張提案。

ジェネレータ-イテレータメソッド

この説ではジェネレータイテレータのメソッドについて説明します。これらはジェネレータ関数の実行制御に使用できます。

以下のジェネレータメソッドの呼び出しは、ジェネレータが既に実行中の場合 `ValueError` 例外を送出する点に注意してください。

`generator.__next__()`

ジェネレータ関数の実行を開始するか、最後に yield 式が実行されたところから再開します。ジェネレータ関数が *__next__()* メソッドによって再開された時、その時点の *yield* 式の値は常に `None` と評価されます。その後次の *yield* 式まで実行し、ジェネレータは一時停止し、*expression_list* の値を *__next__()* メソッドの呼び出し元に返します。ジェネレータが次の値を yield せずに終了した場合、`StopIteration` 例外が送出されます。

このメソッドは通常、例えば *for* ループや組み込みの `next()` 関数によって暗黙に呼び出されます。

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

ジェネレータ関数の内部へ値を ” 送り”、実行を再開します。引数の *value* はその時点の yield 式の結果になります。*send()* メソッドは次にジェネレータが生成した値を返し、ジェネレータが次の値を生成することなく終了すると `StopIteration` を送出します。*send()* が呼び出されてジェネレータが開始するときは、値を受け取る yield 式が存在しないので、`None` を引数として呼び出さなければなりません。

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

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

ジェネレータが中断した位置で例外を発生させて、そのジェネレータ関数が生成する次の値を返します。ジェネレータが値を生成することなく終了すると `StopIteration` が発生します。ジェネレータ関数が渡された例外を捕捉しない、もしくは違う例外を発生させるなら、その例外は呼び出し元へ伝搬されます。

In typical use, this is called with a single exception instance similar to the way the *raise* keyword is used.

For backwards compatibility, however, the second signature is supported, following a convention from older versions of Python. The *type* argument should be an exception class, and *value* should be an exception instance. If the *value* is not provided, the *type* constructor is called to get an instance.

If *traceback* is provided, it is set on the exception, otherwise any existing `__traceback__` attribute stored in *value* may be cleared.

バージョン 3.12 で変更: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

`generator.close()`

Raises a `GeneratorExit` at the point where the generator function was paused. If the generator function catches the exception and returns a value, this value is returned from *close()*. If the generator function is already closed, or raises `GeneratorExit` (by not catching the exception), *close()* returns `None`. If the generator yields a value, a `RuntimeError` is raised. If the generator raises any other exception, it is propagated to the caller. If the generator has already exited due to an exception or normal exit, *close()* returns `None` and has no other effect.

バージョン 3.13 で変更: If a generator returns a value upon being closed, the value is returned by *close()*.

使用例

以下の簡単なサンプルはジェネレータとジェネレータ関数の振る舞いを実際に紹介します:

```
>>> 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.
```

`yield from` の使用例は、”What's New in Python.” の pep-380 を参照してください。

非同期ジェネレータ関数 (asynchronous generator function)

async def を使用して定義された関数やメソッドに yield 式があると、その関数は **非同期ジェネレータ** 関数として定義されます。

非同期ジェネレータ関数が呼び出されると、非同期ジェネレータオブジェクトと呼ばれる非同期イテレータが返されます。そして、そのオブジェクトはジェネレータ関数の実行を制御します。通常、非同期ジェネレータオブジェクトは、コルーチン関数内の *async for* 文で使われ、これはジェネレータオブジェクトが *for* 文で使われる様子に類似します。

非同期ジェネレータのメソッドの 1 つを呼び出すと *awaitable* オブジェクトが返され、このオブジェクトが動く番になったときに実行が開始されます。そのときに実行は最初の yield 式まで進み、そこで再び中断され、*expression_list* の値を待機中のコルーチンに返します。ジェネレータと同様に、中断とは、現在のローカル変数束縛、命令ポインタ、内部評価スタック、および例外処理の状態など、すべてのローカルな状態が保たれることを意味します。非同期ジェネレータのメソッドから次のオブジェクトが返されたことで実行が再開されると、関数はあたかも yield 式が単なる外部呼び出しであるかのように処理を進めていきます。再開後の yield 式の値は、実行を再開したメソッドによって異なります。*__anext__()* を使った場合は、結果は `None` になります。そうではなく、*asend()* が使用された場合は、結果はそのメソッドに渡された値になります。

If an asynchronous generator happens to exit early by *break*, the caller task being cancelled, or other exceptions, the generator's async cleanup code will run and possibly raise exceptions or access context variables in an unexpected context--perhaps after the lifetime of tasks it depends, or during the event loop shutdown when the async-generator garbage collection hook is called. To prevent this, the caller must explicitly close the async generator by calling *aclose()* method to finalize the generator and ultimately detach it from the event loop.

非同期ジェネレータ関数では、*try* 構造内の任意の場所で yield 式が使用できます。ただし、非同期ジェネレータが、(参照カウントがゼロに達するか、ガベージコレクションによる) 終了処理より前に再開されない場合、`try` 構造内の yield 式は失敗となり、実行待ちだった *finally* 節が実行されます。このケースでは、非同期ジェネレータが作動しているイベントループやスケジューラの責務は、非同期ジェネレータの *aclose()* メソッドを呼び出し、残りのコルーチンオブジェクトを実行し、それによって実行待ちだった `finally` 節が実行できるようにします。

To take care of finalization upon event loop termination, an event loop should define a *finalizer* function which takes an asynchronous generator-iterator and presumably calls *aclose()* and executes the coroutine. This *finalizer* may be registered by calling `sys.set_asyncgen_hooks()`. When first iterated over, an asynchronous generator-iterator will store the registered *finalizer* to be called upon finalization. For a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in Lib/asyncio/base_events.py.

`yield from <expr>` 式は、非同期ジェネレータ関数で使われると文法エラーになります。

非同期ジェネレータイテレータメソッド

この小節では、ジェネレータ関数の実行制御に使われる非同期ジェネレータイテレータのメソッドについて説明します。

coroutine agen.**__anext__**()

Returns an awaitable which when run starts to execute the asynchronous generator or resumes it at the last executed yield expression. When an asynchronous generator function is resumed with an *__anext__()* method, the current yield expression always evaluates to `None` in the returned awaitable, which when run will continue to the next yield expression. The value of the *expression_list* of the yield expression is the value of the `StopIteration` exception raised by the completing coroutine. If the asynchronous generator exits without yielding another value, the awaitable instead raises a `StopAsyncIteration` exception, signalling that the asynchronous iteration has completed.

このメソッドは通常、*for* ループによって暗黙に呼び出されます。

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

Returns an awaitable which when run resumes the execution of the asynchronous generator. As with the *send()* method for a generator, this "sends" a value into the asynchronous generator function, and the *value* argument becomes the result of the current yield expression. The awaitable returned by the *asend()* method will return the next value yielded by the generator as the value of the raised `StopIteration`, or raises `StopAsyncIteration` if the asynchronous generator exits without yielding another value. When *asend()* is called to start the asynchronous generator, it must be called with `None` as the argument, because there is no yield expression that could receive the value.

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

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

Returns an awaitable that raises an exception of type `type` at the point where the asynchronous generator was paused, and returns the next value yielded by the generator function as the value of the raised `StopIteration` exception. If the asynchronous generator exits without yielding another value, a `StopAsyncIteration` exception is raised by the awaitable. If the generator function does not catch the passed-in exception, or raises a different exception, then when the awaitable is run that exception propagates to the caller of the awaitable.

バージョン 3.12 で変更: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

coroutine agen.**aclose**()

Returns an awaitable that when run will throw a `GeneratorExit` into the asynchronous generator function at the point where it was paused. If the asynchronous generator function then exits gracefully, is already closed, or raises `GeneratorExit` (by not catching the exception), then the returned awaitable will raise a `StopIteration` exception. Any further awaitables returned by subsequent calls to the asynchronous generator will raise a `StopAsyncIteration` exception. If the asynchronous generator

yields a value, a `RuntimeError` is raised by the awaitable. If the asynchronous generator raises any other exception, it is propagated to the caller of the awaitable. If the asynchronous generator has already exited due to an exception or normal exit, then further calls to *aclose()* will return an awaitable that does nothing.

6.3 プライマリ

プライマリは、言語において最も結合の強い操作を表します。文法は以下のようになります:

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

6.3.1 属性参照

属性参照は、プライマリの後ろにピリオドと名前を連ねたものです:

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

The primary must evaluate to an object of a type that supports attribute references, which most objects do. This object is then asked to produce the attribute whose name is the identifier. The type and value produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

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

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

6.3.2 添字表記 (subscription)

The subscription of an instance of a *container class* will generally select an element from the container. The subscription of a *generic class* will generally return a GenericAlias object.

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

When an object is subscripted, the interpreter will evaluate the primary and the expression list.

The primary must evaluate to an object that supports subscription. An object may support subscription through defining one or both of *__getitem__()* and *__class_getitem__()*. When the primary is subscripted, the evaluated result of the expression list will be passed to one of these methods. For more details on when `__class_getitem__` is called instead of `__getitem__`, see *__class_getitem__ versus __getitem__*.

If the expression list contains at least one comma, it will evaluate to a `tuple` containing the items of the expression list. Otherwise, the expression list will evaluate to the value of the list's sole member.

組み込みオブジェクトでは、*__getitem__()* によって添字表記をサポートするオブジェクトには 2 種類あります:

1. マッピング。プライマリが **マッピング** であれば、式リストの値評価結果はマップ内のいずれかのキー値に相当するオブジェクトにならなければなりません。添字表記は、そのキーに対応するマッピング内の値 (value) を選択します。組み込みのマッピングクラスの例は `dict` クラスです。
2. シーケンス。プライマリが **シーケンス** であれば、式リストの評価結果は `int` または `slice` (以下の節で論じます) でなければなりません。組み込みのシーケンスクラスの例には `str`、`list`、`tuple` クラスが含まれます。

The formal syntax makes no special provision for negative indices in *sequences*. However, built-in sequences all provide a *__getitem__()* method that interprets negative indices by adding the length of the sequence to the index so that, for example, `x[-1]` selects the last item of `x`. The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero). Since the support for negative indices and slicing occurs in the object's *__getitem__()* method, subclasses overriding this method will need to explicitly add that support.

文字列 は文字 (*character*) を要素とする特別な種類のシーケンスです。文字は個別の型ではなく、1 文字だけからなる文字列です。

6.3.3 スライス表記 (slicing)

スライス表記はシーケンスオブジェクト (文字列、タプルまたはリスト) におけるある範囲の要素を選択します。スライス表記は式として用いたり、代入や *del* 文の対象として用いたりできます。スライス表記の構文は以下のようになります:

```
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
```

上記の形式的な構文法にはあいまいなところがあります: 式リストに見えるものは、スライスリストにも見えるため、添字表記はスライス表記としても解釈されうるということです。(スライスリストが適切なスライスを含まない場合)、これ以上の構文の複雑化はせず、スライス表記としての解釈よりも添字表記としての解釈が優先されるように定義することで、あいまいさを取り除いています。

The semantics for a slicing are as follows. The primary is indexed (using the same *__getitem__()* method as normal subscription) with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of a proper slice is a slice object (see section **標準型の階層**) whose `start`, `stop` and `step` attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

6.3.4 呼び出し (call)

呼び出しは、呼び出し可能オブジェクト (例えば *function*) を *arguments* の系列とともに呼び出します。系列は空の系列であってもかまいません:

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

最後の位置引数やキーワード引数の後にカンマをつけてもかまいません。構文の意味付けに影響を及ぼすことはありません。

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and all objects having a *__call__()* method are callable). All argument expressions are evaluated before the call is attempted. Please refer to section **関数定義** for the syntax of formal *parameter* lists.

キーワード引数が存在する場合、以下のようにして最初に位置引数 (positional argument) に変換されます。ま

ず、値の入っていないスロットが仮引数に対して生成されます。N 個の位置引数がある場合、位置引数は先頭の N スロットに配置されます。次に、各キーワード引数について、識別子を使って対応するスロットを決定します (識別子が最初の仮引数名と同じなら、最初のスロットを使う、といった具合です)。スロットがすでにすべて埋まっていたなら TypeError 例外が送出されます。それ以外の場合、引数をスロットに埋めていきます。(式が None であっても、その式でスロットを埋めます)。全ての引数が処理されたら、まだ埋められていないスロットをそれぞれに対応する関数定義時のデフォルト値で埋めます。(デフォルト値は、関数が定義されたときに一度だけ計算されます; 従って、リストや辞書のような変更可能なオブジェクトがデフォルト値として使われると、対応するスロットに引数を指定しない限り、このオブジェクトが全ての呼び出しから共有されます; このような状況は通常避けるべきです。) デフォルト値が指定されていない、値の埋められていないスロットが残っている場合 TypeError 例外が送出されます。そうでない場合、値の埋められたスロットからなるリストが呼び出しの引数として使われます。

CPython 実装の詳細: 実装では、名前を持たない位置引数を受け取る組み込み関数を提供されるかもしれません。そういった引数がドキュメント化のために ' 名付けられて' いたとしても、実際には名付けられていないのでキーワードでは提供されません。CPython では、C 言語で実装された関数の、名前を持たない位置引数をパースするために PyArg_ParseTuple() を使用します。

仮引数スロットの数よりも多くの位置引数がある場合、構文 *identifier を使って指定された仮引数がないかぎり、TypeError 例外が送出されます; 仮引数 *identifier がある場合、この仮引数は余分な位置引数が入ったタプル (もしくは、余分な位置引数がない場合には空のタプル) を受け取ります。

キーワード引数のいずれかが仮引数名に対応しない場合、構文 **identifier を使って指定された仮引数がない限り、TypeError 例外が送出されます; 仮引数 **identifier がある場合、この仮引数は余分なキーワード引数が入った (キーワードをキーとし、引数値をキーに対応する値とした) 辞書を受け取ります。余分なキーワード引数がない場合には、空の (新たな) 辞書を受け取ります。

関数呼び出しに *expression という構文が現れる場合は、expression の評価結果は イテラブル でなければなりません。そのイテラブルの要素は、追加の位置引数であるかのように扱われます。f(x1, x2, *y, x3, x4) という呼び出しにおいて、*y* の評価結果がシーケンス *y1*, ..., *yM* だった場合は、この呼び出しは M+4 個の位置引数 *x1*, *x2*, *y1*, ..., *yM*, *x3*, *x4* での呼び出しと同じになります。

この結論としては、*expression 構文がキーワード引数の **後ろ** に来ることもありますが、キーワード引数 (と任意の **expression 引数 -- 下を参照) よりも **前** にあるものとして処理されます。従って、このような動作になります:

```
>>> 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'
```

(次のページに続く)

(前のページからの続き)

```
>>> f(1, *(2,))
1 2
```

キーワード引数と *expression 構文を同じ呼び出しで一緒に使うことはあまりないので、実際に上記のような混乱が頻繁に生じることはありません。

関数呼び出しで **expression 構文が使われた場合、expression の評価結果は **マッピング** でなければなりません。その内容は追加のキーワード引数として扱われます。キーにマッチする引数が (明示的なキーワード引数によって、あるいは他のアンパックの中で) 既に値を与えられていたなら、TypeError 例外が送出されます。

When **expression is used, each key in this mapping must be a string. Each value from the mapping is assigned to the first formal parameter eligible for keyword assignment whose name is equal to the key. A key need not be a Python identifier (e.g. "max-temp °F" is acceptable, although it will not match any formal parameter that could be declared). If there is no match to a formal parameter the key-value pair is collected by the ** parameter, if there is one, or if there is not, a TypeError exception is raised.

*identifier や **identifier 構文を使った仮引数は、位置引数スロットやキーワード引数名にすることができません。

バージョン 3.5 で変更: 関数呼び出しは任意の数の * アンパックと ** アンパックを受け取り、位置引数はイテラブルアンパック (*) の後ろに置け、キーワード引数は辞書アンパック (**) の後ろに置けるようになりました。最初に **PEP 448** で提案されました。

呼び出しを行うと、例外を送出しない限り、常に何らかの値を返します。None を返す場合もあります。戻り値がどのように算出されるかは、呼び出し可能オブジェクトの形態によって異なります。

各形態では---

ユーザ定義関数: 関数のコードブロックに引数リストが渡され、実行されます。コードブロックは、まず仮引数を実引数に結合 (bind) します; この動作については **関数定義** で記述しています。コードブロックで *return* 文が実行される際に、関数呼び出しの戻り値 (return value) が決定されます。

組み込み関数またはメソッド: 結果はインタプリタに依存します; 組み込み関数やメソッドの詳細は built-in-funcs を参照してください。

クラスオブジェクト: そのクラスの新しいインスタンスが返されます。

クラスインスタンスメソッド: 対応するユーザ定義の関数が呼び出されます。このとき、呼び出し時の引数リストより一つ長い引数リストで呼び出されます: インスタンスが引数リストの先頭に追加されます。

クラスインスタンス:

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

6.4 Await 式

awaitable オブジェクトでの *coroutine* 実行を一時停止します。*coroutine function* 内でのみ使用できます。

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

Added in version 3.5.

6.5 べき乗演算 (power operator)

べき乗演算は、左側にある単項演算子よりも強い結合優先順位となります。一方、右側にある単項演算子よりは弱い結合優先順位になっています。構文は以下のようになります:

```
power   ::=   (await_expr | primary) ["**" u_expr]
```

従って、べき乗演算子と単項演算子からなる演算列が丸括弧で囲われていない場合、演算子は右から左へと評価されます (この場合は演算子の評価順序を強制しません。つまり `-1**2` は `-1` になります)。

べき乗演算子の意味は、二つの引数で呼び出される組み込み関数 `pow()` と同じで、左引数を右引数乗して与えます。数値引数はまず共通の型に変換され、結果はその型です。

整数の被演算子では、第二引数が負でない限り、結果は被演算子と同じ型になります; 第二引数が負の場合、全ての引数は浮動小数点型に変換され、浮動小数点型が返されます。例えば `10**2` は `100` を返しますが、`10**-2` は `0.01` を返します。

`0.0` を負の数でべき乗すると `ZeroDivisionError` を送出します。負の数を小数でべき乗した結果は複素数 (`complex` number) になります。(以前のバージョンでは `ValueError` を送出していました)

This operation can be customized using the special *__pow__()* and *__rpow__()* methods.

6.6 単項算術演算とビット単位演算 (unary arithmetic and bitwise operation)

全ての単項算術演算とビット単位演算は、同じ優先順位を持っています:

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

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

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

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

上記の三つはいずれも、引数が正しい型でない場合には `TypeError` 例外が送出されます。

6.7 二項算術演算 (binary arithmetic operation)

二項算術演算は、慣習的な優先順位を踏襲しています。演算子のいずれかは、特定の非数値型にも適用されるので注意してください。べき乗 (power) 演算子を除き、演算子には二つのレベル、すなわち乗算的 (multiplicatie) 演算子と加算的 (additie) 演算子しかありません:

```
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
```

`*` (乗算: multiplication) 演算子は、引数同士の積を与えます。引数は、両方とも数値であるか、片方が整数で他方がシーケンスかのどちらかでなければなりません。前者の場合、数値は共通の型に変換された後乗算されます。後者の場合、シーケンスの繰り返し操作が行われます。繰り返し数を負にすると、空のシーケンスを与えます。

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

`@` (at) 演算子は行列の乗算に対し使用されます。Python の組み込み型はこの演算子を実装していません。

This operation can be customized using the special *__matmul__()* and *__rmatmul__()* methods.

Added in version 3.5.

`/` (除算: division) および `//` (切り捨て除算: floor division) は、引数同士の商を与えます。数値引数はまず共通の型に変換されます。整数の除算結果は浮動小数点になりますが、整数の切り捨て除算結果は整数になります; この場合、結果は数学的な除算に 'floor' 関数 を適用したものになります。ゼロによる除算を行うと `ZeroDivisionError` 例外を送出します。

The division operation can be customized using the special *__truediv__()* and *__rtruediv__()* methods. The floor division operation can be customized using the special *__floordiv__()* and *__rfloordiv__()* methods.

The % (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating-point numbers, e.g., `3.14%0.7` equals `0.34` (since `3.14` equals `4*0.7 + 0.34`.) The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand[*1].

切り捨て除算演算と剰余演算は、恒等式: `x == (x//y)*y + (x%y)` の関係にあります。切り捨て除算や剰余はまた、組み込み関数 `divmod()`: `divmod(x, y) == (x//y, x%y)` とも関係しています。[*2] 。

% 演算子は、数値に対する剰余演算を行うのに加えて、文字列 (string) オブジェクトにオーバーロードされ、旧式の文字列の書式化 (いわゆる補間) を行います。文字列の書式化の構文は Python ライブラリリファレンス old-string-formatting 節を参照してください。

The *modulo* operation can be customized using the special *__mod__()* and *__rmod__()* methods.

The floor division operator, the modulo operator, and the `divmod()` function are not defined for complex numbers. Instead, convert to a floating-point number using the `abs()` function if appropriate.

+ (加算) 演算は、引数同士の和を与えます。引数は双方とも数値型か、双方とも同じ型のシーケンスでなければなりません。前者の場合、数値は共通の型に変換され、加算されます。後者の場合、シーケンスは結合 (concatenate) されます。

This operation can be customized using the special *__add__()* and *__radd__()* methods.

- (減算) 演算は、引数間で減算を行った値を返します。数値引数はまず共通の型に変換されます。

This operation can be customized using the special *__sub__()* and *__rsub__()* methods.

6.8 シフト演算 (shifting operation)

シフト演算は、算術演算よりも低い優先順位を持っています:

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

これらは整数を引数にとります。引数は共通の型に変換されます。シフト演算は第一引数を、第二引数で与えられたビット数だけ、左または右にビットシフトします。

[*1] `abs(x%y) < abs(y)` は数学的には真となりますが、浮動小数点に対する演算の場合には、値丸め (roundoff) のために数値計算的に真にならない場合があります。例えば、Python の浮動小数点型が IEEE754 倍精度数型になっているプラットフォームを仮定すると、`-1e-100 % 1e100` は `1e100` と同じ符号になるはずなのに、計算結果は `-1e-100 + 1e100` となります。これは数値計算的には厳密に `1e100` と等価です。関数 `math.fmod()` は、最初の引数と符号が一致するような値を返すので、上記の場合には `-1e-100` を返します。どちらのアプローチが適切かは、アプリケーションに依存します。

[*2] x が y の正確な整数倍に非常に近いと、丸めのために `x//y` が `(x-x%y)//y` よりも 1 だけ大きくなる可能性があります。そのような場合、Python は `divmod(x,y)[0] * y + x % y` が x に非常に近くなるという関係を保つために、後者の値を返します。

The left shift operation can be customized using the special *__lshift__()* and *__rlshift__()* methods. The right shift operation can be customized using the special *__rshift__()* and *__rrshift__()* methods.

n ビットの右シフトは `pow(2,n)` による除算として定義されます。n ビットの左シフトは `pow(2,n)` による乗算として定義されます。

6.9 ビット単位演算の二項演算 (binary bitwise operation)

以下の三つのビット単位演算には、それぞれ異なる優先順位レベルがあります:

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

The `&` operator yields the bitwise AND of its arguments, which must be integers or one of them must be a custom object overriding *__and__()* or *__rand__()* special methods.

The `^` operator yields the bitwise XOR (exclusive OR) of its arguments, which must be integers or one of them must be a custom object overriding *__xor__()* or *__rxor__()* special methods.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be integers or one of them must be a custom object overriding *__or__()* or *__ror__()* special methods.

6.10 比較

C 言語と違って、Python における比較演算子は同じ優先順位をもっており、全ての算術演算子、シフト演算子、ビット単位演算子よりも低くなっています。また `a < b < c` が数学で伝統的に用いられているのと同じ解釈になる点も C 言語と違います:

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

Comparisons yield boolean values: `True` or `False`. Custom *rich comparison methods* may return non-boolean values. In this case Python will call `bool()` on such value in boolean contexts.

比較はいくらでも連鎖することができます。例えば `x < y <= z` は `x < y and y <= z` と等価になります。ただしこの場合、前者では `y` はただ一度だけ評価される点が異なります (どちらの場合でも、`x < y` が偽になると `z` の値はまったく評価されません)。

形式的には、*a*, *b*, *c*, ..., *y*, *z* が式で *op1*, *op2*, ..., *opN* が比較演算子である場合、`a op1 b op2 c ... y opN z` は `a op1 b and b op2 c and ... y opN z` と等価になります。ただし、前者では各式は多くても一度しか評価されません。

`a op1 b op2 c` と書いた場合、*a* から *c* までの範囲にあるかどうかのテストを指すのではないことに注意してください。例えば `x < y > z` は (きれいな書き方ではありませんが) 完全に正しい文法です。

6.10.1 値の比較

演算子 `<`, `>`, `==`, `>=`, `<=`, および `!=` は 2 つのオブジェクトの値を比較します。オブジェクトが同じ型を持つ必要はありません。

オブジェクト、値、および型 の章では、オブジェクトは (型や id に加えて) 値を持つことを述べています。オブジェクトの値は Python ではやや抽象的な概念です: 例えば、オブジェクトの値にアクセスする正統な方法はありません。また、その全てのデータ属性から構成されるなどの特定の方法で、オブジェクトの値を構築する必要性もありません。比較演算子は、オブジェクトの値とは何かについての特定の概念を実装しています。この比較の実装によって、間接的にオブジェクトの値を定義していると考えることもできます。

Because all types are (direct or indirect) subtypes of `object`, they inherit the default comparison behavior from `object`. Types can customize their comparison behavior by implementing *rich comparison methods* like *`__lt__()`*, described in **基本的なカスタマイズ**.

等価比較 (`==` および `!=`) のデフォルトの振る舞いは、オブジェクトの同一性に基づいています。従って、同一のインスタンスの等価比較の結果は等しいとなり、同一でないインスタンスの等価比較の結果は等しくないとなります。デフォルトの振る舞いをこのようにしたのは、全てのオブジェクトを反射的 (reflexive つまり `x is y` ならば `x == y`) なものにしたかったからです。

デフォルトの順序比較 (`<`, `>`, `<=`, `>=`) は提供されません; 比較しようとすると `TypeError` が送出されます。この振る舞いをデフォルトの振る舞いにした動機は、等価性と同じような不変性が欠けているからです。

同一でないインスタンスは常に等価でないとする等価比較のデフォルトの振る舞いは、型が必要とするオブジェクトの値や値に基づいた等価性の実用的な定義とは対照的に思えるでしょう。そのような型では比較の振る舞いをカスタマイズする必要が出てきて、実際にたくさんの組み込み型でそれが行われています。

次のリストでは、最重要の組み込み型の比較の振る舞いを解説しています。

- いくつかの組み込みの数値型 (typesnumeric) と標準ライブラリの型 `fractions.Fraction` および `decimal.Decimal` は、これらの型の範囲で異なる型とも比較できますが、複素数では順序比較がサポートされていないという制限があります。関わる型の制限の範囲内では、精度のロス無しに数学的に (アルゴリズム的に) 正しい比較が行われます。

  非数値である `float('NaN')` と `decimal.Decimal('NaN')` は特別です。数と非数値との任意の順序比較は偽です。直観に反する帰結として、非数値は自分自身と等価ではないことになります。例えば `x = float('NaN')` ならば、`3 < x`, `x < 3`, `x == x` は全て偽で、`x != x` は真です。この振る舞いは IEEE 754

に従ったものです。

- None and NotImplemented are singletons. **PEP 8** advises that comparisons for singletons should always be done with is or is not, never the equality operators.

- バイナリシーケンス (bytes または bytearray のインスタンス) は、これらの型の範囲で異なる型とも比較できます。比較は要素の数としての値を使った辞書式順序で行われます。

- 文字列 (str のインスタンス) の比較は、文字の Unicode のコードポイントの数としての値 (組み込み関数 ord() の返り値) を使った辞書式順序で行われます。[*3]

  文字列とバイナリシーケンスは直接には比較できません。

- シーケンス (tuple, list, or range のインスタンス) の比較は、同じ型どうしでしか行えず、range は順序比較をサポートしていません。異なる型どうしの等価比較の結果は等価でないとなり、異なる型どうしの順序比較は TypeError を送出します。

  Sequences compare lexicographically using comparison of corresponding elements. The built-in containers typically assume identical objects are equal to themselves. That lets them bypass equality tests for identical objects to improve performance and to maintain their internal invariants.

  組み込みのコレクションどうしの辞書式比較は次のように動作します:

  – 比較の結果が等価となる 2 つのコレクションは、同じ型、同じ長さ、対応する要素どうしの比較の結果が等価でなければなりません (例えば、[1,2] == (1,2) は型が同じでないので偽です)。

  – 順序比較をサポートしているコレクションの順序は、最初の等価でない要素の順序と同じになります (例えば、[1,2,x] <= [1,2,y] は x <= y と同じ値になります)。対応する要素が存在しない場合、短い方のコレクションの方が先の順序となります (例えば、[1,2] < [1,2,3] は真です)。

- マッピング (dict のインスタンス) の比較の結果が等価となるのは、同じ (key, value) を持っているときかつそのときに限ります。キーと値の等価比較では反射性が強制されます。

  順序比較 (<, >, <=, >=) は TypeError を送出します。

- 集合 (set または frozenset のインスタンス) の比較は、これらの型の範囲で異なる型とも行えます。

  集合には、部分集合あるいは上位集合かどうかを基準とする順序比較が定義されています。この関係は全順

[*3] Unicode 標準では、**コードポイント** *(code point)* (例えば、U+0041) と **抽象文字** *(abstract character)* (例えば、"LATIN CAPITAL LETTER A") を区別します。Unicode のほとんどの抽象文字は 1 つのコードポイントだけを使って表現されますが、複数のコードポイントの列を使っても表現できる抽象文字もたくさんあります。例えば、抽象文字 "LATIN CAPITAL LETTER C WITH CEDILLA" はコード位置 U+00C7 にある **合成済み文字** *(precomposed character)* 1 つだけでも表現できますし、コード位置 U+0043 (LATIN CAPITAL LETTER C) にある **基底文字** *(base character)* の後ろに、コード位置 U+0327 (COMBINING CEDILLA) にある **結合文字** *(combining character)* が続く列としても表現できます。
文字列の比較操作は Unicode のコードポイントのレベルで行われます。これは人間にとっては直感的ではないかもしれません。例えば、"\u00C7" == "\u0043\u0327" は、どちらの文字も同じ抽象文字 "LATIN CAPITAL LETTER C WITH CEDILLA" を表現しているにもかかわらず、その結果は False となります。
抽象文字のレベルで (つまり、人間にとって直感的な方法で) 文字列を比較するには unicodedata.normalize() を使ってください。

序を定義しません (例えば、{1,2} と {2,3} という 2 つの集合は片方がもう一方の部分集合でもなく上位集合でもありません)。従って、集合は全順序性に依存する関数の引数として適切ではありません (例えば、min(), max(), sorted() は集合のリストを入力として与えると未定義な結果となります)。

集合の比較では、その要素の反射性が強制されます。

- 他の組み込み型のほとんどは比較メソッドが実装されておらず、デフォルトの比較の振る舞いを継承します。

比較の振る舞いをカスタマイズしたユーザ定義クラスは、可能なら次の一貫性の規則に従う必要があります:

- 等価比較は反射的でなければなりません。つまり、同一のオブジェクトは等しくなければなりません:

  x is y ならば x == y

- 比較は対称的でなければなりません。つまり、以下の式の結果は同じでなければなりません:

  x == y と y == x

  x != y と y != x

  x < y と y > x

  x <= y と y >= x

- 比較は推移的でなければなりません。以下の (包括的でない) 例がその説明です:

  x > y and y > z ならば x > z

  x < y and y <= z ならば x < z

- 比較の逆は真偽値の否定でなければなりません。つまり、以下の式の結果は同じでなければなりません:

  x == y と not x != y

  x < y と not x >= y (全順序の場合)

  x > y と not x <= y (全順序の場合)

  最後の 2 式は全順序コレクションに当てはまります (たとえばシーケンスには当てはまりますが、集合やマッピングには当てはまりません)。total_ordering() デコレータも参照してください。

- hash() の結果は等価性と一貫している必要があります。等価なオブジェクトどうしは同じハッシュ値を持つか、ハッシュ値が計算できないものとされる必要があります。

Python はこの一貫性規則を強制しません。事実、非数値がこの規則に従わない例となります。

6.10.2 所属検査演算

演算子 *in* および *not in* は所属関係を調べます。`x in s` の評価は、x が s の要素であれば `True` となり、そうでなければ `False` となります。`x not in s` は `x in s` の否定を返します。すべての組み込みのシーケンス型と集合型に加えて、辞書も `in` を辞書が与えられたキーを持っているかを調べる演算子としてサポートしています。リスト、タプル、集合、凍結集合、辞書、collections.deque のようなコンテナ型において、式 `x in y` は `any(x is e or x == e for e in y)` と等価です。

文字列やバイト列型については、`x in y` は x が y の部分文字列であるとき、かつそのときに限り `True` になります。これは `y.find(x) != -1` と等価です。空文字列は、他の任意の文字列の部分文字列とみなされます。従って `"" in "abc"` は `True` を返すことになります。

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

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

Lastly, the old-style iteration protocol is tried: if a class defines *__getitem__()*, `x in y` is `True` if and only if there is a non-negative integer index i such that `x is y[i] or x == y[i]`, and no lower integer index raises the `IndexError` exception. (If any other exception is raised, it is as if *in* raised that exception).

演算子 *not in* は *in* の真理値を反転した値として定義されています。

6.10.3 同一性の比較

演算子 *is* および *is not* は、オブジェクトの同一性に対するテストを行います: `x is y` は、x と y が同じオブジェクトを指すとき、かつそのときに限り真になります。オブジェクトの同一性は `id()` 関数を使って判定されます。`x is not y` は *is* の真値を反転したものになります。[*4]

6.11 ブール演算 (boolean operation)

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

[*4] 自動的なガベージコレクション、フリーリスト、ディスクリプタの動的特性のために、インスタンスメソッドや定数の比較を行うようなときに *is* 演算子の利用は、一見すると普通ではない振る舞いだと気付くかもしれません。詳細はそれぞれのドキュメントを確認してください。

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: `False`, `None`, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. User-defined objects can customize their truth value by providing a *`__bool__()`* method.

演算子 *`not`* は、引数が偽である場合には `True` を、それ以外の場合には `False` になります。

式 `x and y` は、まず x を評価します; x が偽なら x の値を返します; それ以外の場合には、y を評価した結果値を返します。

式 `x or y` は、まず x を評価します; x が真なら x の値を返します; それ以外の場合には、y を評価した結果値を返します。

なお、*`and`* も *`or`* も、返す値を `True` や `False` に制限せず、最後に評価した引数を返します。この仕様が便利なときもあります。例えば `s` が文字列で、空文字列ならデフォルトの値に置き換えたいとき、式 `s or 'foo'` は望んだ値を与えます。*`not`* は必ず新しい値を作成するので、引数の型に関係なくブール値を返します (例えば、`not 'foo'` は `''` ではなく `False` になります)。

6.12 代入式

```
assignment_expression  ::=  [identifier ":="] expression
```

An assignment expression (sometimes also called a "named expression" or "walrus") assigns an ***expression*** to an ***identifier***, while also returning the value of the ***expression***.

One common use case is when handling matched regular expressions:

```
if matching := pattern.search(data):
    do_something(matching)
```

Or, when processing a file stream in chunks:

```
while chunk := file.read(9000):
    process(chunk)
```

Assignment expressions must be surrounded by parentheses when used as expression statements and when used as sub-expressions in slicing, conditional, lambda, keyword-argument, and comprehension-if expressions and in `assert`, `with`, and `assignment` statements. In all other places where they can be used, parentheses are not required, including in `if` and `while` statements.

Added in version 3.8: 代入式に関してより詳しくは **PEP 572** を参照してください。

6.13 条件式 (Conditional Expressions)

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

条件式 (しばしば ”三項演算子” と呼ばれます) は最も優先度が低い Python の演算です。

`x if C else y` という式は最初に x ではなく条件 C を評価します。C が true の場合 x が評価され値が返されます。それ以外の場合には y が評価され返されます。

条件演算に関してより詳しくは **PEP 308** を参照してください。

6.14 ラムダ (lambda)

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

ラムダ式 (ラムダ形式とも呼ばれます) は無名関数を作成するのに使います。式 `lambda parameters: expression` は関数オブジェクトになります。この無名オブジェクトは以下に定義されている関数オブジェクト同様に動作します:

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

引数の一覧の構文は **関数定義** を参照してください。ラムダ式で作成された関数は文やアノテーションを含むことができない点に注意してください。

6.15 式のリスト

```
expression_list     ::=  expression ("," expression)* [","]
starred_list        ::=  starred_item ("," starred_item)* [","]
starred_expression  ::=  expression | (starred_item ",")* [starred_item]
starred_item        ::=  assignment_expression | "*" or_expr
```

リスト表示や辞書表示の一部になっているものを除き、少なくとも一つのカンマを含む式のリストはタプルになります。タプルの長さは、リストにある式の数に等しくなります。式は左から右へ評価されます。

アスタリスク * は **イテラブルのアンパック** を意味します。この被演算子は イテラブル でなければなりません。このイテラブルはアンパックされた位置で要素のシーケンスに展開され、新しいタプル、リスト、集合に入れ込まれます。

Added in version 3.5: 式リストでのイテラブルのアンパックは最初に **PEP 448** で提案されました。

A trailing comma is required only to create a one-item tuple, such as `1,`; it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: `()`.)

6.16 評価順序

Python は、式を左から右へと順に評価します。ただし、代入式を評価するときは、右辺が左辺よりも先に評価されます。

以下に示す実行文の各行での評価順序は、添え字の数字順序と同じになります:

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

6.17 演算子の優先順位

以下の表は Python における演算子の優先順位を要約したものです。優先順位の最も高い (結合が最も強い) ものから最も低い (結合が最も弱い) ものに並べてあります。同じボックス内の演算子の優先順位は同じです。構文が明示的に示されていないものは二項演算子です。同じボックス内の演算子は、左から右へとグループ化されます (例外として、べき乗および条件式は右から左にグループ化されます)。

比較 節で述べられているように、比較、所属、同一性のテストは全てが同じ優先順位を持っていて、左から右に連鎖するという特徴を持っていることに注意してください。

演算子	説明
(expressions...), [expressions...], {key: value...}, {expressions...}	結合式または括弧式、リスト表示、辞書表示、集合表示
x[index], x[index:index], x(arguments...), x.attribute	添字指定、スライス操作、呼び出し、属性参照
await x	Await 式
**	べき乗[*5]
+x, -x, ~x	正数、負数、ビット単位 NOT
*, @, /, //, %	乗算、行列乗算、除算、切り捨て除算、剰余[*6]
+, -	加算および減算
<<, >>	シフト演算
&	ビット単位 AND
^	ビット単位 XOR
|	ビット単位 OR
in, *not in*, *is*, *is not*, <, <=, >, >=, !=, ==	所属や同一性のテストを含む比較
not x	ブール演算 NOT
and	ブール演算 AND
or	ブール演算 OR
if -- else	条件式
lambda	ラムダ式
:=	代入式

脚注

[*5] べき乗演算子 ** は、右側にある単項算術演算子あるいは単項ビット演算子より弱い結合優先順位となります。つまり 2**-1 は 0.5 になります。

[*6] % 演算子は文字列フォーマットにも使われ、同じ優先順位が当てはまります。

第

SEVEN

単純文 (SIMPLE STATEMENT)

単純文とは、単一の論理行内に収められる文です。単一の行内には、複数の単純文をセミコロンで区切って入れることができます。単純文の構文は以下の通りです:

```
simple_stmt  ::=  expression_stmt
                | assert_stmt
                | assignment_stmt
                | augmented_assignment_stmt
                | annotated_assignment_stmt
                | pass_stmt
                | del_stmt
                | return_stmt
                | yield_stmt
                | raise_stmt
                | break_stmt
                | continue_stmt
                | import_stmt
                | future_stmt
                | global_stmt
                | nonlocal_stmt
                | type_stmt
```

7.1 式文 (expression statement)

式文は、(主に対話的な使い方では) 値を計算して出力するために使ったり、(通常は) プロシジャ (procedure: 有意な結果を返さない関数のことです; Python では、プロシジャは値 None を返します) を呼び出すために使います。その他の使い方でも式文を使うことができますし、有用なこともあります。式文の構文は以下の通りです:

```
expression_stmt   ::=   starred_expression
```

式文は式のリスト (単一の式のこともあります) を値評価します。

対話モードでは、値が None でなければ、値を組み込み関数 repr() で文字列に変換して、その結果の文字列を標準出力に一行使って書き出します。(None になる式文の値は書き出されないので、プロシジャの呼び出しを行っても出力は得られません。)

7.2 代入文 (assignment statement)

代入文は、名前を値に (再) 束縛したり、変更可能なオブジェクトの属性や要素を変更したりするために使われます:

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

(*attributeref*, *subscription*, *slicing* の構文については プライマリ 節を参照してください。)

代入文は式のリスト (これは単一の式でも、カンマで区切られた式リストでもよく、後者はタプルになることを思い出してください) を評価し、得られたそれぞれのオブジェクトをターゲット (target) のリストに対して左から右へと代入してゆきます。

代入はターゲット (リスト) の形式に従って再帰的に行われます。ターゲットが変更可能なオブジェクト (属性参照、添字表記、またはスライス) の一部である場合、この変更可能なオブジェクトは最終的に代入を実行して、その代入が有効な操作であるか判断しなければなりません。代入が不可能な場合には例外を発行することもできま

す。型ごとにみられる規則や、送出される例外は、そのオブジェクト型定義で与えられています (**標準型の階層** 節を参照してください).

ターゲットリストは、丸括弧や角括弧で囲まれていてもよく、それに対するオブジェクトの代入は、以下のように再帰的に定義されています。

- ターゲットリストのターゲットが 1 つだけでコンマが続いておらず、任意に丸括弧で囲われている場合、オブジェクトはそのターゲットに代入されます。
- その他:
  - " 星付き" のターゲットと呼ばれる、頭にアスタリスクが一つ付いたターゲットがターゲットリストに一つだけ含まれている場合: オブジェクトはイテラブルで、少なくともターゲットリストのターゲットの数よりも一つ少ない要素を持たなければなりません。星付きのターゲットより前のターゲットに、イテラブルの先頭の要素が左から右へ代入されます。星付きのターゲットより後ろのターゲットに、イテラブルの末尾の要素が代入されます。星付きのターゲットに、イテラブルの残った要素のリストが代入されます (リスト空でもかまいません)。
  - そうでない場合: オブジェクトは、ターゲットリストのターゲットと同じ数の要素を持つイテラブルでなければならず、要素は左から右へ対応するターゲットに代入されます。

単一のターゲットへの単一のオブジェクトの代入は、以下のようにして再帰的に定義されています。

- ターゲットが識別子 (名前) の場合:
  - 名前が現在のコードブロック内の *global* や *nonlocal* 文に書かれていないければ: 名前は現在のローカル名前空間内のオブジェクトに束縛されます。
  - そうでなければ: 名前はそれぞれグローバル名前空間内か、*nonlocal* で決められた外側の名前空間内のオブジェクトに束縛されます。

  名前がすでに束縛済みの場合、再束縛 (rebind) がおこなわれます。再束縛によって、以前その名前に束縛されていたオブジェクトの参照カウント (reference count) がゼロになった場合、オブジェクトは解放 (deallocate) され、デストラクタ (destructor) が (存在すれば) 呼び出されます。
- ターゲットが属性参照の場合: 参照されている一次語の式が値評価されます。値は代入可能な属性を伴うオブジェクトでなければなりません; そうでなければ、`TypeError` が送出されます。次に、このオブジェクトに対して、被代入オブジェクトを指定した属性に代入してよいか問い合わせます; 代入を実行できない場合、例外 (通常は `AttributeError` ですが、必然ではありません) を送出します。

  注意: オブジェクトがクラスインスタンスで、代入演算子の両辺に属性参照があるとき、右辺式の `a.x` はインスタンスの属性と (インスタンスの属性が存在しなければ) クラス属性のどちらにもアクセスする可能性があります。左辺のターゲット `a.x` は常にインスタンスの属性として割り当てられ、必要ならば生成されます。このとおり、現れる二つの `a.x` は同じ値を参照するとは限りません: 右辺式はクラス属性を参照し、左辺は新しいインスタンス属性を代入のターゲットとして生成するようなとき:

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

このことは、property() で作成されたプロパティのようなデスクリプタ属性に対しては、必ずしもあてはまるとは限りません。

- ターゲットが添字表記なら: 参照されている一次語式が評価されます。参照から (リストのような) ミュータブルなシーケンスオブジェクトか、(辞書のような) マッピングオブジェクトが得られなければなりません。次に、添字表記の表す式が評価されます。

  一次語が (リストのような) ミュータブルなシーケンスオブジェクトであれば、添字表記は整数を与えなければなりません。整数が負なら、シーケンスの長さが加算されます。整数は最終的に、シーケンスの長さよりも小さな非負の整数でなくてはならず、シーケンスは、そのインデクスに持つ要素に被代入オブジェクトを代入してよいか問い合わせられます。インデクスが範囲外なら、IndexError が送出されます (添字指定されたシーケンスに代入を行っても、リスト要素の新たな追加はできません)。

  If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/value pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

  For user-defined objects, the *__setitem__()* method is called with appropriate arguments.

- ターゲットがスライスなら: 参照されている一次語式が評価されます。一次語式は、(リストのような) ミュータブルなシーケンスオブジェクトを与えなければなりません。被代入オブジェクトは同じ型のシーケンスオブジェクトでなければなりません。次に、スライスの下限と上限を示す式があれば評価されます; デフォルト値はそれぞれ 0 とシーケンスの長さです。上限と下限の評価は整数でなければなりません。いずれかの境界が負数なら、シーケンスの長さが加算されます。最終的に、境界は 0 からシーケンスの長さまでに収まるように刈りこまれます。最後に、スライスを被代入オブジェクトで置き換えてよいかシーケンスオブジェクトに問い合わせます。ターゲットシーケンスで許されている限り、スライスの長さは被代入シーケンスの長さと異なっていてよく、この場合にはターゲットシーケンスの長さが変更されます。

CPython 実装の詳細: 現在の実装では、ターゲットの構文は式の構文と同じであるとみなされており、無効な構文はコード生成フェーズ中に詳細なエラーメッセージを伴って拒否されます。

代入の定義によれば、左辺と右辺のオーバーラップは ' 同時 (simultaneous)' です (例えば a, b = b, a は二つの変数を入れ替えます) が、代入対象となる変数群 **どうし** のオーバーラップは左から右へ起こり、混乱の元です。例えば、以下のプログラムは [0, 2] を出力してしまいます:

```
x = [0, 1]
i = 0
i, x[i] = 1, 2         # i is updated, then x[i] is updated
print(x)
```

参考

PEP 3132 - Extended Iterable Unpacking
*target の指定機能。

7.2.1 累算代入文 (augmented assignment statement)

累算代入文は、二項演算と代入文を組み合わせて一つの文にしたものです:

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

(最後の 3 つの構文定義については プライマリ を参照してください。)

累算代入文は、ターゲット (通常の代入文と違って、アンパックは起こりません) と式リストを評価し、それら二つの被演算子間で特定の累算代入型の二項演算を行い、結果をもとのターゲットに代入します。ターゲットは一度しか評価されません。

An augmented assignment statement like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

通常の代入とは違い、累算代入文は右辺を評価する **前に** 左辺を評価します。たとえば、`a[i] += f(x)` はまず `a[i]` を調べ、`f(x)` を評価して加算を行い、最後に結果を `a[i]` に割り当てます。

累算代入文で行われる代入は、タプルへの代入や、一文中に複数のターゲットが存在する場合を除き、通常の代入と同じように扱われます。同様に、累算代入で行われる二項演算は、場合によって **インプレース演算** が行われることを除き、通常の二項演算と同じです。

属性参照のターゲットの場合、クラス属性とインスタンス属性についての注意 と同様に通常の代入が適用されます。

7.2.2 注釈付き代入文 (annotated assignment statements)

注釈 代入は、1 つの文の中で変数や属性のアノテーションとオプションの代入文を組み合わせたものです:

```
annotated_assignment_stmt  ::=  augtarget ":" expression
                                ["=" (starred_expression | yield_expression)]
```

通常の 代入文 *(assignment statement)* との違いは、代入先が 1 つに限定されることだけです。

The assignment target is considered ”simple” if it consists of a single name that is not enclosed in parentheses. For simple assignment targets, if in class or module scope, the annotations are evaluated and stored in a special class or module attribute `__annotations__` that is a dictionary mapping from variable names (mangled if private) to evaluated annotations. This attribute is writable and is automatically created at the start of class or module body execution, if annotations are found statically.

If the assignment target is not simple (an attribute, subscript node, or parenthesized name), the annotation is evaluated if in class or module scope, but not stored.

関数スコープで名前に注釈が付いていた場合は、その名前はその関数スコープでローカルなものになります。注釈は絶対に評価されず、関数スコープにも格納されません。

If the right hand side is present, an annotated assignment performs the actual assignment before evaluating annotations (where applicable). If the right hand side is not present for an expression target, then the interpreter evaluates the target except for the last *__setitem__()* or *__setattr__()* call.

参考

PEP 526 - Syntax for Variable Annotations
: (クラス変数やインスタンス変数を含んだ) 変数の型注釈を付ける、コメントで表現するのではない文法の追加提案。

PEP 484 - Type hints
: `typing` モジュールを追加し、静的解析ツールや IDE で使える型アノテーションの標準的な文法を提供する提案。

バージョン 3.8 で変更: Now annotated assignments allow the same expressions in the right hand side as regular assignments. Previously, some expressions (like un-parenthesized tuple expressions) caused a syntax error.

7.3 assert 文

assert 文は、プログラム内にデバッグ用アサーション (debugging assertion) を仕掛けるための便利な方法です:

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

単純な形式 `assert expression` は

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

と等価です。拡張形式 `assert expression1, expression2` は、これと等価です

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

上記の等価関係は、`__debug__` と `AssertionError` が、同名の組み込み変数を参照しているという前提の上に成り立っています。現在の実装では、組み込み変数 `__debug__` は通常の状況では `True` であり、最適化が要求された場合（コマンドラインオプション `-O`）は `False` です。現状のコード生成器は、コンパイル時に最適化が要求されていると assert 文のコードを一切出力しません。実行に失敗した式のソースコードをエラーメッセージ内に入れる必要はありません; コードはスタックトレース内で表示されます。

`__debug__` への代入は不正な操作です。組み込み変数の値は、インタプリタが開始するときに決定されます。

7.4 pass 文

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

pass はヌル操作 (null operation) です --- *pass* が実行されても、何も起きません。*pass* は、構文法的には文が必要だが、コードとしては何も実行したくない場合のプレースホルダとして有用です。例えば:

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

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

7.5 del 文

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

オブジェクトの削除 (deletion) は、代入の定義と非常に似た方法で再帰的に定義されています。ここでは完全な詳細は記述せず、いくつかのヒントを述べるにとどめます。

ターゲットリストに対する削除は、各々のターゲットを左から右へと順に再帰的に削除します。

名前の削除は、ローカルまたはグローバル名前空間からその名前の束縛を取り除きます。どちらの名前空間かは、名前が同じコードブロック内の *global* 文で宣言されているかどうかによります。名前が未束縛 (unbound) なら、NameError 例外が送出されます。

属性参照、添字表記、およびスライスの削除操作は、対象となる一次語オブジェクトに渡されます; スライスの削除は一般的には適切な型の空のスライスを代入するのと等価です (が、この仕様自体もスライスされるオブジェクトで決定されています)。

バージョン 3.2 で変更: 以前は、ある名前がネストしたブロックの自由変数として表れる場合は、ローカル名前空間からその名前を削除することは不正な処理でした。

7.6 return 文

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

return は、関数定義内で構文法的にネストして現れますが、ネストしたクラス定義内には現れません。

式リストがある場合、リストが値評価されます。それ以外の場合は None で置き換えられます。

return を使うと、式リスト (または None) を戻り値として、現在の関数呼び出しから抜け出します。

return によって、*finally* 節をともなう *try* 文の外に処理が引き渡されると、実際に関数から抜ける前に finally 節が実行されます。

ジェネレータ関数では、*return* 文はジェネレータの終わりを示し、StopIteration 例外を送出させます。返された値は (あれば)、StopIteration を構成する引数に使われ、StopIteration.value 属性になります。

非同期ジェネレータ関数では、引数無しの *return* 文は非同期ジェネレータの終わりを示し、StopAsyncIteration を送出させます。引数ありの return 文は、非同期ジェネレータ関数では文法エラーです。

7.7 yield 文

```
yield_stmt  ::=  yield_expression
```

yield 文は意味的に *yield* **式** と同じです。yield 文を用いると yield 式文で必要な括弧を省略することが出来ます。例えば、yield 文

```
yield <expr>
yield from <expr>
```

は以下の yield 式文と等価です

```
(yield <expr>)
(yield from <expr>)
```

yield 式及び文は *generator* を定義するときに、その本体内でのみ使うことが出来ます。関数定義内で yield を使用することで、その定義は通常の関数でなくジェネレータ関数になります。

yield の意味の完全な説明は、*Yield* **式** 節を参照してください。

7.8 raise 文

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

If no expressions are present, *raise* re-raises the exception that is currently being handled, which is also known as the *active exception*. If there isn't currently an active exception, a `RuntimeError` exception is raised indicating that this is an error.

そうでなければ、*raise* は最初の式を、例外オブジェクトとして評価します。これは、`BaseException` のサブクラスまたはインスタンスでなければなりません。クラスなら、例外インスタンスが必要なとき、クラスを無引数でインスタンス化することで得られます。

例外の **型** は例外インスタンスのクラスで、**値** はインスタンスそのものです。

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

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

The `from` clause is used for exception chaining: if given, the second *expression* must be another exception class or instance. If the second expression is an exception instance, it will be attached to the raised exception as the `__cause__` attribute (which is writable). If the expression is an exception class, the class will be instantiated and the resulting exception instance will be attached to the raised exception as the `__cause__` attribute. If the raised exception is not handled, both exceptions will be printed:

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

The above exception was the direct cause of the following exception:

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

A similar mechanism works implicitly if a new exception is raised when an exception is already being handled. An exception may be handled when an *except* or *finally* clause, or a *with* statement, is used. The previous exception is then attached as the new exception's `__context__` attribute:

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

During handling of the above exception, another exception occurred:

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

(次のページに続く)

(前のページからの続き)

```
RuntimeError: Something bad happened
```

Exception chaining can be explicitly suppressed by specifying `None` in the `from` clause:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

例外に関する追加情報は **例外** 節にあります。また、例外処理に関する情報は *try* **文** 節にあります。

バージョン 3.3 で変更: `None` が `raise X from Y` の `Y` として使えるようになりました。

Added the `__suppress_context__` attribute to suppress automatic display of the exception context.

バージョン 3.11 で変更: If the traceback of the active exception is modified in an *except* clause, a subsequent `raise` statement re-raises the exception with the modified traceback. Previously, the exception was re-raised with the traceback it had when it was caught.

7.9 break 文

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

break 文は、構文としては *for* ループや *while* ループの内側でのみ出現することができますが、ループ内の関数定義やクラス定義の内側には出現できません。

break 文は、文を囲う最も内側のループを終了させ、ループにオプションの `else` 節がある場合にはそれをスキップします。

for ループを *break* によって終了すると、ループ制御ターゲットはその時の値を保持します。

break が *finally* 節を伴う *try* 文の外側に処理を渡す際には、ループを実際に抜ける前にその `finally` 節が実行されます。

7.10 continue 文

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

continue 文は *for* ループや *while* ループ内のネストで構文法的にのみ現れますが、ループ内の関数定義やクラス定義の中には現れません。*continue* 文は、文を囲う最も内側のループの次の周期に処理を継続します。

continue が *finally* 句を持った *try* 文を抜けるとき、その finally 句が次のループサイクルを始める前に実行されます。

7.11 import 文

```
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" relative_module "import" "*"
module            ::=   (identifier ".")* identifier
relative_module   ::=   "."* module | "."+
```

(*from* 節が無い) 基本の import 文は 2 つのステップで実行されます:

1. モジュールを見付け出し、必要であればロードし初期化する
2. *import* 文が表れるスコープのローカル名前空間で名前を定義する。

文が (カンマで区切られた) 複数の節を含んでいるときは、ちょうどその節が個別の import 文に分割されたかのように、2 つのステップが節ごとに個別に実行されます。

モジュールを見付け、ロードする 1 つ目のステップの詳細については、**インポートシステム** の節により詳しく書かれています。そこでは、インポートシステムの動作をカスタマイズするのに使える全てのフックの仕組みだけでなく、様々な種類のインポートできるパッケージとモジュールについても解説されています。このステップが失敗するということは、おそらくモジュールが見付からないか、**あるいは** モジュールにあるコードの実行を含め、モジュールの初期化の途中でエラーが起きるかのどちらかが起きていることに注意してください。

要求したモジュールが無事に取得できた場合、次の 3 つのうちの 1 つの方法でローカル名前空間で使えるようになります:

- モジュール名の後に as が続いていた場合は、as の後ろの名前を直接、インポートされたモジュールが束縛します。
- 他の名前が指定されておらず、インポートされているモジュールが最上位のモジュールだった場合、そのモジュール名がインポートされたモジュールへの参照として、ローカル名前空間で束縛されます
- インポートされているモジュールが最上位のモジュール **でない** 場合、モジュールを含む最上位のパッケージ名が、そのパッケージへの参照として、ローカル名前空間で束縛されます。インポートされたモジュールには、直接ではなく完全修飾名を使ってアクセスしなければなりません

from 形式ではもう少し複雑な手順を踏みます:

1. *from* 節で指定されたモジュールを見付け出し、必要であればロードし初期化する;
2. *import* 節で指定されたそれぞれの識別子に対し以下の処理を行う:
    1. インポートされたモジュールがその識別子名の属性を持っているかを確認する
    2. その識別子名の属性を持っていなかった場合は、その識別子名でサブモジュールのインポートを試み、インポートされたモジュールにその属性があるか再度確認する
    3. 属性が見付からない場合は、ImportError を送出する。
    4. 属性が見付かった場合は、as 節があるならそこの名前、そうでないなら属性名を使って、その値への参照がローカル名前空間に保存される

例:

```
import foo                 # foo imported and bound locally
import foo.bar.baz         # foo, foo.bar, and foo.bar.baz imported, foo bound locally
import foo.bar.baz as fbb  # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz bound as fbb
from foo.bar import baz    # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz bound as baz
from foo import attr       # foo imported and foo.attr bound as attr
```

識別子のリストが星 ('*') に置き換わっている場合は、モジュールで定義されている公開された全ての名前が、*import* 文がいるスコープのローカル名前空間に束縛されます。

モジュールで定義される **公開された名前** は、モジュールの名前空間にある __all__ という名前の変数を調べることで決定されます; その変数が定義されている場合は、それはモジュールで定義されたかインポートされた名前からなる、文字列のシーケンスでなければいけません。__all__ で列挙された名前は、全て公開されていると見なされ、存在することが要求されます。__all__ が定義されていない場合、公開された名前とは、モジュールの名前空間で見付かった、アンダースコア文字 ('_') で始まらない全ての名前のことです。__all__ は全ての公開 API を含むべきです。これは API の一部でないもの (そのモジュールでインポートされ使われているライブラリモジュールなど) をうっかり外部に公開してしまわないための仕組みです。

インポートのワイルドカード形式 --- from module import * --- は、モジュールレベルでのみ許されます。クラスや関数定義でこの形式を使おうとすると、SyntaxError が送出されます。

インポートするモジュールを指定するとき、そのモジュールの絶対名 (absolute name) を指定する必要はありません。モジュールやパッケージが他のパッケージに含まれている場合、共通のトップパッケージからそのパッケージ名を記述することなく相対インポートすることができます。*from* の後に指定されるモジュールやパッケージの先頭に複数個のドットを付けることで、正確な名前を指定することなしに現在のパッケージ階層からいくつ上の階層へ行くかを指定することができます。先頭のドットが 1 つの場合、import をおこなっているモジュールが存在する現在のパッケージを示します。3 つのドットは 2 つ上のレベルを示します。なので、pkg パッケージの中のモジュールで `from . import mod` を実行すると、pkg.mod をインポートすることになります。pkg.subpkg1 の中から `from ..subpkg2 import mod` を実行すると、pkg.subpkg2.mod をインポートします。相対インポートの仕様は *Package Relative Imports* の節に含まれています。

どのモジュールがロードされるべきかを動的に決めたいアプリケーションのために、組み込み関数 `importlib.import_module()` が提供されています。

Raises an auditing event `import` with arguments `module`, `filename`, `sys.path`, `sys.meta_path`, `sys.path_hooks`.

7.11.1 future 文 (future statement)

future **文** は、将来の特定の新たな機能が標準化された Python のリリースで利用可能になるような構文や意味付けを使って、特定のモジュールをコンパイルさせるための、コンパイラに対する指示句 (directive) です。

future 文は互換性のない変更がされた将来の Python のバージョンに容易に移行するためのものです。future 文によって新機能が標準となるリリースの前にそれをモジュール単位で使用することが出来ます。

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

future 文は、モジュールの先頭周辺に書かなければなりません。future 文の前に書いてよい内容は以下です：

- モジュールのドキュメンテーション文字列 (あれば)
- コメント ,
- 空行 ,
- その他の future 文。

future 文を使う必要がある唯一の機能は `annotations` です (**PEP 563** を参照してください)。

future 文で有効にできる歴史的な機能は、今でも Python 3 が認識します。そのリスト

は absolute_import, division, generator_stop, generators, unicode_literals, print_function, nested_scopes, with_statement です。これらは既に全てが有効になっていて、後方互換性のためだけに残されているため、冗長なだけです。

future 文は、コンパイル時に特別なやり方で認識され、扱われます: 言語の中核をなす構文構成 (construct) に対する意味付けが変更されている場合、変更部分はしばしば異なるコードを生成することで実現されています。新たな機能によって、(新たな予約語のような) 互換性のない新たな構文が取り入れられることさえあります。この場合、コンパイラはモジュールを別のやりかたで解析する必要があるかもしれません。こうしたコード生成に関する決定は、実行時まで先延ばしすることはできません。

これまでの全てのリリースにおいて、コンパイラはどの機能が定義済みかを知っており、future 文に未知の機能が含まれている場合にはコンパイル時エラーを送出します。

future 文の実行時における直接的な意味付けは、import 文と同じです。標準モジュール __future__ があり、これについては後で述べます。__future__ は、future 文が実行される際に通常の方法で import されます。

future 文の実行時における特別な意味付けは、future 文で有効化される特定の機能によって変わります。

以下の文には、何ら特殊な意味はないので注意してください:

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

これは future 文ではありません; この文は通常の import 文であり、その他の特殊な意味付けや構文的な制限はありません。

Code compiled by calls to the built-in functions exec() and compile() that occur in a module M containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can be controlled by optional arguments to compile() --- see the documentation of that function for details.

対話的インタプリタのプロンプトでタイプ入力した future 文は、その後のインタプリタセッション中で有効になります。インタプリタを -i オプションで起動して実行すべきスクリプト名を渡し、スクリプト中に future 文を入れておくと、新たな機能はスクリプトが実行された後に開始する対話セッションで有効になります。

参考

PEP 236 - Back to the ___future___
___future___ 機構の原案

7.12 global 文

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

global 文は、現在のコードブロック全体で維持される宣言文です。*global* 文は、列挙した識別子をグローバル変数として解釈するよう指定することを意味します。global を使わずにグローバル変数に代入を行うことは不可能ですが、自由変数を使えばその変数をグローバルであると宣言せずにグローバル変数を参照することができます。

global 文で列挙する名前は、同じコードブロック中で、プログラムテキスト上 global 文より前に使ってはなりません。

global 文で列挙する名前は、仮引数として、または *with* 文や *except* 節のターゲットとして定義されたり、*for* のターゲットリスト、*class* 定義、関数定義、*import* 文、変数アノテーションの中で定義されたりしてはなりません。

CPython 実装の詳細: 現在の実装では、これらの制限のうち幾つかについては強制していませんが、プログラムでこの緩和された仕様を乱用すべきではありません。将来の実装では、この制限を強制したり、暗黙のうちにプログラムの意味付けを変更したりする可能性があります。

プログラマのための注意点: *global* はパーザに対する指示句 (directive) です。この指示句は、global 文と同時に読み込まれたコードに対してのみ適用されます。特に、組み込みの exec() 関数内に入っている global 文は、関数の呼び出しを **含んでいる** コードブロック内に効果を及ぼすことはなく、そのような文字列に含まれているコードは、関数の呼び出しを含むコード内の global 文に影響を受けません。同様のことが、関数 eval() および compile() にも当てはまります。

7.13 nonlocal 文

```
nonlocal_stmt  ::=  "nonlocal" identifier ("," identifier)*
```

When the definition of a function or class is nested (enclosed) within the definitions of other functions, its nonlocal scopes are the local scopes of the enclosing functions. The *nonlocal* statement causes the listed identifiers to refer to names previously bound in nonlocal scopes. It allows encapsulated code to rebind such nonlocal identifiers. If a name is bound in more than one nonlocal scope, the nearest binding is used. If a name is not bound in any nonlocal scope, or if there is no nonlocal scope, a SyntaxError is raised.

The nonlocal statement applies to the entire scope of a function or class body. A SyntaxError is raised if a variable is used or assigned to prior to its nonlocal declaration in the scope.

参考

PEP 3104 - Access to Names in Outer Scopes
nonlocal 文の詳細。

Programmer's note: *nonlocal* is a directive to the parser and applies only to code parsed along with it. See the note for the *global* statement.

7.14 The type statement

```
type_stmt  ::=  'type' identifier [type_params] "=" expression
```

The type statement declares a type alias, which is an instance of typing.TypeAliasType.

For example, the following statement creates a type alias:

```
type Point = tuple[float, float]
```

This code is roughly equivalent to:

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

annotation-def indicates an *annotation scope*, which behaves mostly like a function, but with several small differences.

The value of the type alias is evaluated in the annotation scope. It is not evaluated when the type alias is created, but only when the value is accessed through the type alias's __value__ attribute (see *Lazy evaluation*). This allows the type alias to refer to names that are not yet defined.

Type aliases may be made generic by adding a *type parameter list* after the name. See *Generic type aliases* for more.

type is a *soft keyword*.

Added in version 3.12.

参考

PEP 695 - Type Parameter Syntax

Introduced the `type` statement and syntax for generic classes and functions.

第

EIGHT

複合文 (COMPOUND STATEMENT)

複合文には、他の文 (のグループ) が入ります; 複合文は、中に入っている他の文の実行の制御に何らかのやり方で影響を及ぼします。一般的には、複合文は複数行にまたがって書かれますが、全部の文を一行に連ねた単純な書き方もあります。

if 、*while* 、および *for* 文は、伝統的な制御フロー構成を実現します。*try* は例外処理および/または一連の文に対するクリーンアップコードを指定します。それに対して、*with* 文はコードのかたまりの前後でコードの初期化と終了処理を実行できるようにします。関数とクラス定義もまた、構文的には複合文です。

複合文は、一つ以上の ' 節 (clause)' からなります。節は、ヘッダと ' スイート (suite)' からなります。一つの複合文を成す各節のヘッダは、全て同じインデントレベルに置かれます。各節のヘッダは一意に識別するキーワードで始まり、コロンで終わります。スイートは、節によって制御される文の集まりです。スイートは、ヘッダがある行のコロンの後にセミコロンで区切って置かれた一つ以上の単純文、または、ヘッダに続く行で一つ多くインデントされた文の集まりです。後者の形式のスイートに限り、さらに複合文をネストできます; 以下の文は、*else* 節がどちらの *if* 節に属するかがはっきりしないなどの理由から不正になります:

```
if test1: if test2: print(x)
```

また、このコンテキスト中では、セミコロンによる結合はコロンより強いです。従って、以下の例では、print() の呼び出しはは全て実行されるか、全く実行されないかのどちらかです:

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

まとめると、以下のようになります:

```
compound_stmt   ::=   if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | with_stmt
                    | match_stmt
```

```
                    | funcdef
                    | classdef
                    | async_with_stmt
                    | async_for_stmt
                    | async_funcdef
suite           ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement       ::=  stmt_list NEWLINE | compound_stmt
stmt_list       ::=  simple_stmt (";" simple_stmt)* [";"]
```

なお、文は常に NEWLINE か、その後に DEDENT が続いたもので終了します。また、オプションの継続節は必ず、文を開始できない予約語で始まるので、曖昧さは存在しません。(Python では、' ぶら下がり (dangling) *else*' 問題は、ネストされた *if* 文をインデントさせることで解決されます)。

以下の節における文法規則の記述方式は、明確さのために、各節を別々の行に書くようにしています。

8.1 if 文

if 文は、条件分岐を実行するために使われます:

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

if 文は、式を一つ一つ評価してゆき、真になるまで続けて、真になった節のスイートだけを選択します (真: true と偽: false の定義については、**ブール演算** *(boolean operation)* 節を参照してください); 次に、選択したスイートを実行します (そして、*if* 文の他の部分は、実行や評価をされません)。全ての式が偽になった場合、*else* 節があれば、そのスイートが実行されます。

8.2 while 文

while 文は、式の値が真である間、実行を繰り返すために使われます:

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

while 文は式を繰り返し真偽評価し、真であれば最初のスイートを実行します。式が偽であれば (最初から偽になっていることもありえます)、else 節がある場合にはそれを実行し、ループを終了します。

最初のスイート内で *break* 文が実行されると、else 節のスイートを実行することなくループを終了します。*continue* 文が最初のスイート内で実行されると、スイート内にある残りの文の実行をスキップして、式の真偽評価に戻ります。

8.3 for 文

for 文は、シーケンス (文字列、タプルまたはリスト) や、その他の反復可能なオブジェクト (iterable object) 内の要素に渡って反復処理を行うために使われます:

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

starred_list 式は一度だけ評価され、評価結果として **イテラブル** オブジェクトを返す必要があります。そのイテラブルから **イテレータ** が作られます。まず、イテレータから生成される最初の要素が、通常の代入式のルールに従って target_list に代入され (**代入文** *(assignment statement)* 節参照)、1 つ目のスイート (suite) が実行されます。これが、イテレータから生成される各要素に対して繰り返されます。イテレータのすべての要素が処理されたあと、else 節がもしあれば実行され、ループ処理が終了します。

最初のスイートの中で *break* 文が実行されると、else 節のスイートを実行することなくループを終了します。*continue* 文が最初のスイート内で実行されると、スイート内にある残りの文の実行をスキップして、次の要素の処理に移るか、これ以上次の要素が無い場合は else 節の処理に移ります。

for ループはターゲットリスト内の変数への代入を行います。これにより、for ループ内も含めて、それ以前の全ての代入は上書きされます:

```
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
```

ループが終了してもターゲットリスト内の名前は削除されませんが、イテラブルが空の場合には、ループでの代入は全く行われません。ヒント: 組み込み型 range() は、整数の不変算術列を表します。例えば、range(3) を反復すると 0、1 そして 2 の順に結果を返します。

バージョン 3.11 で変更: 式のリストの中でアスタリスク付きの式 (starred elements) を指定できるようになりました。

8.4 try 文

try 文は、ひとまとめの文に対して、例外処理および/またはクリーンアップコードを指定します:

```
try_stmt   ::=  try1_stmt | try2_stmt | try3_stmt
try1_stmt  ::=  "try" ":" suite
                ("except" [expression ["as" identifier]] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try2_stmt  ::=  "try" ":" suite
                ("except" "*" expression ["as" identifier] ":" suite)+
                ["else" ":" suite]
                ["finally" ":" suite]
try3_stmt  ::=  "try" ":" suite
                "finally" ":" suite
```

例外に関するその他の情報は **例外** 節にあります。また、*raise* 文の使用による例外の生成に関する情報は、*raise* **文** 節にあります。

8.4.1 except 節

The except clause(s) specify one or more exception handlers. When no exception occurs in the *try* clause, no exception handler is executed. When an exception occurs in the try suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception.

For an except clause with an expression, the expression must evaluate to an exception type or a tuple of exception types. The raised exception matches an except clause whose expression evaluates to the class or a *non-virtual base class* of the exception object, or to a tuple that contains such a class.

例外がどの except 節にも合致しなかった場合、現在のコードを囲うさらに外側、そして呼び出しスタックへと検索を続けます。[*1]

except 節のヘッダにある式を値評価するときに例外が発生すると、元々のハンドラ検索はキャンセルされ、新たな例外に対する例外ハンドラの検索を現在の *except* 節の外側のコードや呼び出しスタックに対して行います (*try* 文全体が例外を発行したかのように扱われます)。

対応する except 節が見つかると、except 節のスイートが実行されます。その際、as キーワードが except 節

[*1] 例外は、別の例外を送出するような *finally* 節が無い場合にのみ呼び出しスタックへ伝わります。新しい例外によって、古い例外は失われます。

に存在すれば、その後で指定されているターゲットに例外が代入されます。全ての except 節は実行可能なブロックを持っていなければなりません。このブロックの末尾に到達すると、通常は *try* 文全体の直後から実行を継続します。(このことは、ネストされた二つの例外ハンドラが同じ例外に対して存在し、内側のハンドラ内の try 節で例外が発生した場合、外側のハンドラはその例外を処理しないことを意味します。)

例外が as target を使って代入されたとき、それは except 節の終わりに消去されます。これはちょうど、以下のコード:

```
except E as N:
    foo
```

が、以下のコードに翻訳されたかのようなものです:

```
except E as N:
    try:
        foo
    finally:
        del N
```

よって、例外を except 節以降で参照できるようにするためには、別の名前に代入されなければなりません。例外が削除されるのは、トレースバックが付与されると、そのスタックフレームと循環参照を形作り、次のガベージ収集までそのフレーム内のすべての局所変数を生存させてしまうからです。

except 節のスイートが実行される前に、例外が sys モジュールに格納されます。except 節の中では、sys.exception() を呼び出す事によってこの例外にアクセスすることができます。例外ハンドラを抜けると、sys モジュールに格納されている例外の値が、一つ前の値に戻ります:

```
>>> print(sys.exception())
None
>>> try:
...     raise TypeError
... except:
...     print(repr(sys.exception()))
...     try:
...         raise ValueError
...     except:
...         print(repr(sys.exception()))
...     print(repr(sys.exception()))
...
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None
```

8.4.2 except* 節

The except* clause(s) are used for handling ExceptionGroups. The exception type for matching is interpreted as in the case of *except*, but in the case of exception groups we can have partial matches when the type matches some of the exceptions in the group. This means that multiple except* clauses can execute, each handling part of the exception group. Each clause executes at most once and handles an exception group of all matching exceptions. Each exception in the group is handled by at most one except* clause, the first that matches it.

```
>>> try:
...     raise ExceptionGroup("eg",
...         [ValueError(1), TypeError(2), OSError(3), OSError(4)])
... except* TypeError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
... except* OSError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
...
caught <class 'ExceptionGroup'> with nested (TypeError(2),)
caught <class 'ExceptionGroup'> with nested (OSError(3), OSError(4))
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: eg
  +-+---------------- 1 ----------------
    | ValueError: 1
    +------------------------------------
```

Any remaining exceptions that were not handled by any except* clause are re-raised at the end, along with all exceptions that were raised from within the except* clauses. If this list contains more than one exception to reraise, they are combined into an exception group.

If the raised exception is not an exception group and its type matches one of the except* clauses, it is caught and wrapped by an exception group with an empty message string.

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

An except* clause must have a matching expression; it cannot be except*:. Furthermore, this expression cannot contain exception group types, because that would have ambiguous semantics.

It is not possible to mix *except* and except* in the same *try*. *break*, *continue* and *return* cannot appear in an except* clause.

8.4.3 else 節

オプションの else 節は、コントロールフローが *try* スイートを抜け、例外が送出されず、*return* 文、*continue* 文、*break* 文のいずれもが実行されなかった場合に実行されます。else 節で起きた例外は、手前にある *except* 節では処理されません。

8.4.4 finally 節

finally 節がある場合は、'後始末' の対処を指定します。まず *except* 節や *else* 節を含め、*try* 節が実行されます。それらの節の中で例外が起き、その例外が処理されていない場合には、例外は一時的に保存されます。次に finally 節が実行されます。保存された例外があった場合は、finally 節の末尾で再送出されます。finally 節で別の例外が送出される場合は、保存されていた例外は新しい例外のコンテキストとして設定されます。finally 節で *return* 文、*break* 文あるいは *continue* 文を実行した場合は、保存された例外は破棄されます:

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

finally 節を実行している間は、プログラムからは例外情報は利用できません。

try...finally 文の *try* スイート内で *return* 、*break* 、または *continue* 文が実行された場合、finally 節も、この文を '抜け出る途中に' 実行されます。

関数の返り値は最後に実行された *return* 文によって決まります。finally 節は必ず実行されるため、finally 節で実行された return 文は常に最後に実行されることになります:

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

バージョン 3.8 で変更: Python3.8 以前では、実装上の問題により finally 節での *continue* 文は不正でした。

8.5 with 文

with 文は、ブロックの実行を、コンテキストマネージャによって定義されたメソッドでラップするために使われます (*with* 文とコンテキストマネージャ セクションを参照してください)。これにより、よくある *try*...*except*...*finally* 利用パターンをカプセル化して便利に再利用することができます。

```
with_stmt           ::=  "with" ( "(" with_stmt_contents ","? ")" | with_stmt_contents ) ":" suite
with_stmt_contents  ::=  with_item ("," with_item)*
with_item           ::=  expression ["as" target]
```

一つの ” 要素” を持つ *with* 文の実行は以下のように進行します:

1. コンテキスト式 (*with_item* で与えられた式) を評価することで、コンテキストマネージャを取得します。
2. コンテキストマネージャの *__enter__()* メソッドが、後で使うためにロードされます。
3. コンテキストマネージャの *__exit__()* メソッドが、後で使うためにロードされます。
4. コンテキストマネージャの *__enter__()* メソッドが呼ばれます。
5. *with* 文にターゲットが含まれていたら、それに *__enter__()* からの戻り値が代入されます。

   注釈

   with 文は、*__enter__()* メソッドがエラーなく終了した場合には *__exit__()* が常に呼ばれることを保証します。ですので、もしターゲットリストへの代入中にエラーが発生した場合には、これはそのスイートの中で発生したエラーと同じように扱われます。以下のステップ 7 を参照してください。

6. スイートが実行されます。
7. コンテキストマネージャの *__exit__()* メソッドが呼ばれます。スイートが例外によって終了されたのなら、その例外の型、値、トレースバックが *__exit__()* に引数として渡されます。そうでなければ、3 つの None 引数が与えられます。

   スイートが例外により終了され、*__exit__()* メソッドからの戻り値が偽（false）ならば、例外が再送出されます。この戻り値が真（true）ならば例外は抑制され、実行は *with* 文の次の文から続きます。

   もしそのスイートが例外でない何らかの理由で終了した場合、その *__exit__()* からの戻り値は無視されて、実行は発生した終了の種類に応じた通常の位置から継続します。

以下のコード:

```
with EXPRESSION as TARGET:
    SUITE
```

これは次と等価です:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

複数の要素があるとき、コンテキストマネージャは複数の *with* 文がネストされたかのように進行します:

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

これは次と等価です:

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

括弧で囲むことにより、複数のコンテキストマネージャを複数行に渡って書くことができます。例:

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

バージョン 3.1 で変更: 複数のコンテキスト式をサポートしました。

バージョン 3.10 で変更: 括弧で囲むことで、文を複数行に分割して書けるようになりました。

参考

PEP 343 - "with" **ステートメント**
Python の *with* 文の仕様、背景、および例が記載されています。

8.6 match 文

Added in version 3.10.

match 文はパターンマッチングを行う目的で使われます。構文:

```
match_stmt   ::=  'match' subject_expr ":" NEWLINE INDENT case_block+ DEDENT
subject_expr ::=  star_named_expression "," star_named_expressions?
                  | named_expression
case_block   ::=  'case' patterns [guard] ":" block
```

注釈

このセクションでは、一重引用符で囲まれているものは ソフトキーワード を表します。

パターンマッチングは、パターン (case の後ろ) とマッチング対象の値 (match の後ろ) を入力とします。パターン (サブパターンを含みうる) は、マッチング対象の値に対して、マッチするかどうかの判定が行われます。結果として次のことが起こります:

- マッチ成功、もしくはマッチ失敗 (パターン成功・失敗とも呼ばれます)。
- マッチした値の名前への束縛。必要な条件は後述します。

match と case キーワードは ソフトキーワード です。

参考

- **PEP 634** -- 構造的パターンマッチ: 仕様
- **PEP 636** -- 構造的パターンマッチ: チュートリアル

8.6.1 概要

match 文の論理的な動作の流れの概要は次の通りです:

1. サブジェクト式 subject_expr が評価され、サブジェクト値が得られます。サブジェクト式がコンマを含む場合、通常のルール に従ってタプルが作成されます。
2. case_block 内の各パターンに対して、サブジェクト値がマッチするかどうかをチェックします。マッチ成功・失敗の具体的なルールは後述します。マッチングのチェックにより、パターン内の名前の一部あるいはすべてに値が束縛されます。具体的な束縛ルールはパターンの種類によって異なるため、後述します。**マッ**

チが成功したパターンの中で束縛された名前は、そのパターンのブロック内だけでなく、match 文の後でも使用することができます。

> **注釈**
>
> パターンマッチが全体として失敗しても、その中に含まれるサブパターンが成功する可能性があります。失敗したマッチで発生する名前束縛を前提としたコードを書かないように気をつけてください。逆に、マッチが失敗したあとで変数の値が変わっていないというのも前提にしないでください。実際どういう振る舞いになるかは Python の実装依存であり、実装間で異なる可能性があります。色々な実装が最適化を行えるよう、意図的に実装依存としています。

3. パターンが成功した場合、該当のガードが (もしあれば) 評価されます。この場合、パターン内の名前がすべて束縛されていることが保証されています。

   - ガードの評価値が真であるか、もしくはガードがなければ、`case_block` 内の `block` が実行されます。
   - そうでなければ、次の `case_block` に対して再び上記の処理が実行されます。
   - これ以上 case block が存在しない場合は、match 文が終了します。

> **注釈**
>
> 基本的に、match 文のパターンが評価されるという前提でコードを書くべきではありません。インタープリタの実装によっては、結果をキャッシュするなどの最適化を行い、評価をスキップする可能性があります。

簡単な match 文の例:

```
>>> flag = False
>>> match (100, 200):
...     case (100, 300):  # Mismatch: 200 != 300
...         print('Case 1')
...     case (100, 200) if flag:  # Successful match, but guard fails
...         print('Case 2')
...     case (100, y):  # Matches and binds y to 200
...         print(f'Case 3, y: {y}')
...     case _:  # Pattern not attempted
...         print('Case 4, I match anything!')
...
Case 3, y: 200
```

この例では、`if flag` がガードです。ガードについては次のセクションを参照してください。

8.6.2 ガード

```
guard   ::=   "if" named_expression
```

guard (case 一部として現れる) が成功してはじめて、その case ブロックのコードが実行されます。ガードは *if* の後に式を書く形で表記されます。

guard 付きの case ブロックの処理の流れは次のとおりです:

1. case ブロックのパターンが成功するかどうかをチェックする。失敗した場合は、guard は評価されず、次の case ブロックのチェックに進む。
2. パターンが成功した場合は、guard の評価が行われます。
   - guard 条件の評価値が真である場合、該当の case ブロックが選択されます。
   - guard 条件の評価値が偽の場合、該当の case ブロックは選択されません。
   - guard の評価中に例外が送出された場合は、その例外がそのまま送出されます。

ガードは式であるため、副作用を起こすことができます。ガードの評価は、最初の case ブロックから順に、パターンが失敗した case ブロックは飛ばしつつ、一つづつ評価されなければいけません (つまり、ガードの評価は書かれている順番で実行される必要があります)。また、case ブロックが選択された時点で、ガードの評価をそれ以上行ってはいけません。

8.6.3 論駁不可能なケースブロック

論駁不可能なケースブロックとは、何にでもマッチするケースブロックのことです。match 文の中で、論駁不可能なケースブロックは最大一つまで、かつ最後に位置する必要があります。

ケースブロックが論駁不可能であるためには、ガードがなく、パターンが論駁不可能である必要があります。パターンが論駁不可能であるためには、その文法上の構造のみから、それが常に成功することが証明できる必要があります。論駁不可能なパターンは以下のようなもののみです:

- 左辺が論駁不可能である *AS* パターン
- 含まれるパターンのうち少なくとも一つが論駁不可能である *OR* パターン
- キャプチャパターン
- ワイルドカードパターン
- 括弧で囲われた、論駁不可能なパターン

8.6.4 パターン

注釈

このセクションでは、通常の EBNF を拡張した文法記法を使用します。

- SEP.RULE+ という表記は RULE (SEP RULE)* の略です。
- !RULE は否定先読みの条件を表します。

patterns のトップレベルの構文は以下の通りです:

```
patterns        ::=  open_sequence_pattern | pattern
pattern         ::=  as_pattern | or_pattern
closed_pattern  ::=  | literal_pattern
                     | capture_pattern
                     | wildcard_pattern
                     | value_pattern
                     | group_pattern
                     | sequence_pattern
                     | mapping_pattern
                     | class_pattern
```

以下の説明では分かりやすさのため、パターンの振る舞いを簡単に言い表した場合の説明を「簡単に言うと」の後に書いています (そのほとんどは、Raymond Hettinger 氏のドキュメントに影響を受けてのものです)。ただし、これはあくまで理解を助けるためのであり、内部的な実装を必ずしも反映したものでは **ありません** 。また、使用可能なすべてのパターン構造を網羅しているわけではありません。

OR パターン

OR パターンは、縦線 | で区切られた複数のパターンからなります。構文:

```
or_pattern  ::=  "|".closed_pattern+
```

最後のサブパターン以外、**論駁不可能** であってはいけません。また、曖昧さ回避のため、各サブパターンが束縛する名前の組み合わせは、すべて同じである必要があります。

OR パターンでは、サブジェクト値に対して順に各サブパターンのマッチングが行われます。マッチが成功するとそこで終了し、この OR パターンは成功したとみなされます。一方、どのサブパターンも成功しなければ、この

OR パターンは失敗したことになります。

簡単に言うと、P1 | P2 | ... というパターンは P1 をマッチしようとし、失敗すれば P2 を試します。いずれかのパターンがマッチが成功すれば直ちに成功となり、それ以外の場合は失敗となります。

AS パターン

AS パターンはサブジェクト値に対して、*as* キーワードの左側にある OR パターンをマッチさせます。構文:

```
as_pattern   ::=    or_pattern "as" capture_pattern
```

OR パターンが失敗すれば、この AS パターンは失敗となります。成功すれば、サブジェクト値が as キーワードの右側の名前に束縛され、この AS パターンは成功となります。capture_pattern として _ を指定することはできません。

簡単に言うと、P as NAME は P をマッチさせ、成功した場合に NAME = <subject> の代入を行います。

リテラルパターン

リテラルパターンは、一部を除く Python の **リテラル** に対応します。構文:

```
literal_pattern   ::=    signed_number
                       | signed_number "+" NUMBER
                       | signed_number "-" NUMBER
                       | strings
                       | "None"
                       | "True"
                       | "False"
signed_number     ::=    ["-"] NUMBER
```

strings というルールと NUMBER というトークンは *Python* **の文法仕様** で定義されています。クォート 3 つで囲われた文字列や raw 文字列、raw バイト列も使用可能です。*f-strings* は使用できません。

signed_number '+' NUMBER と signed_number '-' NUMBER という構文は **複素数** を表現するためのものです。そのため、左側には実数、右側には虚数を書く必要があります。例: 3 + 4j 。

簡単に言うと、LITERAL は <subject> == LITERAL であるときのみ成功するパターンです。シングルトンである None と True 、False は *is* 演算子を使って比較されます。

キャプチャパターン

キャプチャパターンは、サブジェクト値を名前に束縛します。構文:

```
capture_pattern  ::=  !'_' NAME
```

アンダースコア一文字の _ はキャプチャパターンではありません (!'_' が表しているのはこの条件です) 。*wildcard_pattern* として扱われます。

パターン一つの中で、一つの名前は一度しか束縛することができません。例えば、case x, x: ... は間違いですが、case [x] | x: ... は正しいです。

キャプチャパターンは常に成功します。束縛された名前のスコープは、**PEP 572** で確立された代入式演算子のスコープルールと同じです。すなわち、当てはまる *global* 文 か *nonlocal* 文がない限り、その局所変数のスコープは、該当の match 文を包む最も内側の関数となります。

簡単に言うと、NAME は常に成功し、NAME = <subject> の代入が行われます。

ワイルドカードパターン

ワイルドカードパターンは常に成功 (何に対してもマッチする) し、名前の束縛はしません。構文:

```
wildcard_pattern  ::=  '_'
```

_ は、パターンの中で使用された場合は常に **ソフトキーワード** です。しかし、パターンの中でない場合はソフトキーワードではありません。たとえサブジェクト式や guard 、case ブロックの中でも、通常の変数となります。

簡単に言うと、_ は常に成功します。

値パターン

値パターンは Python で名前の付けられた値を表します。構文:

```
value_pattern  ::=  attr
attr           ::=  name_or_attr "." NAME
name_or_attr   ::=  attr | NAME
```

ドットがついたこの名前は、Python 標準の **名前解決ルール** によって解決されます。このパターンは、解決された値がサブジェクト値と等しい (比較演算子 == に基づく) ときに成功となります。

簡単に言うと、NAME1.NAME2 は <subject> == NAME1.NAME2 であるときのみ成功します。

注釈

一つの match 文で同じ値が複数回出現する場合は、インタープリタが最初に解決された値をキャッシュし、名前解決を何度も行うことなく値を再利用する可能性があります。このキャッシュは、その match 文のその実行一回の間だけで再利用されます。

グループパターン

A group pattern allows users to add parentheses around patterns to emphasize the intended grouping. Otherwise, it has no additional syntax. Syntax:

```
group_pattern  ::=  "(" pattern ")"
```

In simple terms `(P)` has the same effect as `P`.

シーケンスパターン

A sequence pattern contains several subpatterns to be matched against sequence elements. The syntax is similar to the unpacking of a list or tuple.

```
sequence_pattern         ::=  "[" [maybe_sequence_pattern] "]"
                              | "(" [open_sequence_pattern] ")"
open_sequence_pattern    ::=  maybe_star_pattern "," [maybe_sequence_pattern]
maybe_sequence_pattern   ::=  ",".maybe_star_pattern+ ","?
maybe_star_pattern       ::=  star_pattern | pattern
star_pattern             ::=  "*" (capture_pattern | wildcard_pattern)
```

There is no difference if parentheses or square brackets are used for sequence patterns (i.e. `(...)` vs `[...]`).

注釈

A single pattern enclosed in parentheses without a trailing comma (e.g. `(3 | 4)`) is a *group pattern*. While a single pattern enclosed in square brackets (e.g. `[3 | 4]`) is still a sequence pattern.

At most one star subpattern may be in a sequence pattern. The star subpattern may occur in any position.

If no star subpattern is present, the sequence pattern is a fixed-length sequence pattern; otherwise it is a variable-length sequence pattern.

The following is the logical flow for matching a sequence pattern against a subject value:

1. If the subject value is not a sequence[*2], the sequence pattern fails.
2. If the subject value is an instance of `str`, `bytes` or `bytearray` the sequence pattern fails.
3. The subsequent steps depend on whether the sequence pattern is fixed or variable-length.

   If the sequence pattern is fixed-length:

   1. If the length of the subject sequence is not equal to the number of subpatterns, the sequence pattern fails
   2. Subpatterns in the sequence pattern are matched to their corresponding items in the subject sequence from left to right. Matching stops as soon as a subpattern fails. If all subpatterns succeed in matching their corresponding item, the sequence pattern succeeds.

   Otherwise, if the sequence pattern is variable-length:

   1. If the length of the subject sequence is less than the number of non-star subpatterns, the sequence pattern fails.
   2. The leading non-star subpatterns are matched to their corresponding items as for fixed-length sequences.
   3. If the previous step succeeds, the star subpattern matches a list formed of the remaining subject items, excluding the remaining items corresponding to non-star subpatterns following the star subpattern.

[*2] In pattern matching, a sequence is defined as one of the following:

- `collections.abc.Sequence` を継承したクラス。
- `collections.abc.Sequence` として登録された Python クラス。
- a builtin class that has its (CPython) `Py_TPFLAGS_SEQUENCE` bit set
- a class that inherits from any of the above

  The following standard library classes are sequences:

- `array.array`
- `collections.deque`
- `list`
- `memoryview`
- `range`
- `tuple`

注釈

Subject values of type `str`, `bytes`, and `bytearray` do not match sequence patterns.

4. Remaining non-star subpatterns are matched to their corresponding subject items, as for a fixed-length sequence.

> **注釈**
>
> The length of the subject sequence is obtained via `len()` (i.e. via the `__len__()` protocol). This length may be cached by the interpreter in a similar manner as *value patterns*.

In simple terms `[P1, P2, P3, ... , P<N>]` matches only if all the following happens:

- `<subject>` がシーケンスかをチェックする
- `len(subject) == <N>`
- `P1` matches `<subject>[0]` (note that this match can also bind names)
- `P2` matches `<subject>[1]` (note that this match can also bind names)
- ... and so on for the corresponding pattern/element.

マッピングパターン

A mapping pattern contains one or more key-value patterns. The syntax is similar to the construction of a dictionary. Syntax:

```
mapping_pattern      ::=  "{" [items_pattern] "}"
items_pattern        ::=  ",".key_value_pattern+ ","?
key_value_pattern    ::=  (literal_pattern | value_pattern) ":" pattern
                        | double_star_pattern
double_star_pattern  ::=  "**" capture_pattern
```

At most one double star pattern may be in a mapping pattern. The double star pattern must be the last subpattern in the mapping pattern.

Duplicate keys in mapping patterns are disallowed. Duplicate literal keys will raise a `SyntaxError`. Two keys that otherwise have the same value will raise a `ValueError` at runtime.

The following is the logical flow for matching a mapping pattern against a subject value:

1. If the subject value is not a mapping[*3],the mapping pattern fails.

[*3] In pattern matching, a mapping is defined as one of the following:

- a class that inherits from `collections.abc.Mapping`
- a Python class that has been registered as `collections.abc.Mapping`

2. If every key given in the mapping pattern is present in the subject mapping, and the pattern for each key matches the corresponding item of the subject mapping, the mapping pattern succeeds.

3. If duplicate keys are detected in the mapping pattern, the pattern is considered invalid. A `SyntaxError` is raised for duplicate literal values; or a `ValueError` for named keys of the same value.

注釈

Key-value pairs are matched using the two-argument form of the mapping subject's `get()` method. Matched key-value pairs must already be present in the mapping, and not created on-the-fly via `__missing__()` or *`__getitem__()`*.

In simple terms `{KEY1: P1, KEY2: P2, ... }` matches only if all the following happens:

- `<subject>` がマッピングかをチェックする
- `KEY1 in <subject>`
- `P1` は `<subject>[KEY1]` にマッチする
- ... and so on for the corresponding KEY/pattern pair.

クラスパターン

A class pattern represents a class and its positional and keyword arguments (if any). Syntax:

```
class_pattern       ::=  name_or_attr "(" [pattern_arguments ","?] ")"
pattern_arguments   ::=  positional_patterns ["," keyword_patterns]
                         | keyword_patterns
positional_patterns ::=  ",".pattern+
keyword_patterns    ::=  ",".keyword_pattern+
keyword_pattern     ::=  NAME "=" pattern
```

The same keyword should not be repeated in class patterns.

The following is the logical flow for matching a class pattern against a subject value:

1. If `name_or_attr` is not an instance of the builtin `type` , raise `TypeError`.

- a builtin class that has its (CPython) `Py_TPFLAGS_MAPPING` bit set
- a class that inherits from any of the above

The standard library classes `dict` and `types.MappingProxyType` are mappings.

2. If the subject value is not an instance of `name_or_attr` (tested via `isinstance()`), the class pattern fails.

3. If no pattern arguments are present, the pattern succeeds. Otherwise, the subsequent steps depend on whether keyword or positional argument patterns are present.

   For a number of built-in types (specified below), a single positional subpattern is accepted which will match the entire subject; for these types keyword patterns also work as for other types.

   If only keyword patterns are present, they are processed as follows, one by one:

   I. The keyword is looked up as an attribute on the subject.

   - If this raises an exception other than `AttributeError`, the exception bubbles up.
   - If this raises `AttributeError`, the class pattern has failed.
   - Else, the subpattern associated with the keyword pattern is matched against the subject's attribute value. If this fails, the class pattern fails; if this succeeds, the match proceeds to the next keyword.

   II. If all keyword patterns succeed, the class pattern succeeds.

   If any positional patterns are present, they are converted to keyword patterns using the *`__match_args__`* attribute on the class `name_or_attr` before matching:

   I. The equivalent of `getattr(cls, "__match_args__", ())` is called.

   - If this raises an exception, the exception bubbles up.
   - If the returned value is not a tuple, the conversion fails and `TypeError` is raised.
   - If there are more positional patterns than `len(cls.__match_args__)`, `TypeError` is raised.
   - Otherwise, positional pattern `i` is converted to a keyword pattern using `__match_args__[i]` as the keyword. `__match_args__[i]` must be a string; if not `TypeError` is raised.
   - If there are duplicate keywords, `TypeError` is raised.

   参考

   クラスパターンマッチの位置引数のカスタマイズ

   II. Once all positional patterns have been converted to keyword patterns,
   the match proceeds as if there were only keyword patterns.

   For the following built-in types the handling of positional subpatterns is different:

   - `bool`

- bytearray
- bytes
- dict
- float
- frozenset
- int
- list
- set
- str
- tuple

These classes accept a single positional argument, and the pattern there is matched against the whole object rather than an attribute. For example `int(0|1)` matches the value `0`, but not the value `0.0`.

In simple terms `CLS(P1, attr=P2)` matches only if the following happens:

- `isinstance(<subject>, CLS)`
- convert `P1` to a keyword pattern using `CLS.__match_args__`
- For each keyword argument `attr=P2`:
  - `hasattr(<subject>, "attr")`
  - `P2` は `<subject>.attr` にマッチする
- ... and so on for the corresponding keyword argument/pattern pair.

参考

- **PEP 634** -- 構造的パターンマッチ: 仕様
- **PEP 636** -- 構造的パターンマッチ: チュートリアル

8.7 関数定義

関数定義は、ユーザ定義関数オブジェクトを定義します (**標準型の階層** 節参照):

```
funcdef                 ::=  [decorators] "def" funcname [type_params] "(" [parameter_list] ")"
                             ["->" expression] ":" suite
decorators              ::=  decorator+
decorator               ::=  "@" assignment_expression NEWLINE
parameter_list          ::=  defparameter ("," defparameter)* "," "/" ["," [parameter_list_no_poso
                             | parameter_list_no_posonly
parameter_list_no_posonly ::=  defparameter ("," defparameter)* ["," [parameter_list_starargs]]
                             | parameter_list_starargs
parameter_list_starargs ::=  "*" [parameter] ("," defparameter)* ["," ["**" parameter [","]]]
                             | "**" parameter [","]
parameter               ::=  identifier [":" expression]
defparameter            ::=  parameter ["=" expression]
funcname                ::=  identifier
```

関数定義は実行可能な文です。関数定義を実行すると、現在のローカルな名前空間内で関数名を関数オブジェクト (関数の実行可能コードをくるむラッパー) に束縛します。この関数オブジェクトには、関数が呼び出された際に使われるグローバルな名前空間として、現在のグローバルな名前空間への参照が入っています。

関数定義は関数本体を実行しません; 関数本体は関数が呼び出された時にのみ実行されます。[*4]

関数定義は一つ以上の **デコレータ** 式でラップできます。デコレータ式は関数を定義するとき、関数定義の入っているスコープで評価されます。その結果は、関数オブジェクトを唯一の引数にとる呼び出し可能オブジェクトでなければなりません。関数オブジェクトの代わりに、返された値が関数名に束縛されます。複数のデコレータはネストして適用されます。例えば、以下のようなコード:

```
@f1(arg)
@f2
def func(): pass
```

は、だいたい次と等価です

```
def func(): pass
func = f1(arg)(f2(func))
```

ただし、前者のコードでは元々の関数を `func` という名前へ一時的に束縛することはない、というところを除きます。

[*4] A string literal appearing as the first statement in the function body is transformed into the function's *__doc__* attribute and therefore the function's *docstring*.

バージョン 3.9 で変更: Functions may be decorated with any valid *assignment_expression*. Previously, the grammar was much more restrictive; see **PEP 614** for details.

A list of *type parameters* may be given in square brackets between the function's name and the opening parenthesis for its parameter list. This indicates to static type checkers that the function is generic. At runtime, the type parameters can be retrieved from the function's *__type_params__* attribute. See *Generic functions* for more.

バージョン 3.12 で変更: Type parameter lists are new in Python 3.12.

1 つ以上の **仮引数** が *parameter* = *expression* の形を取っているとき、関数は " デフォルト引数値" を持つと言います。デフォルト値を持つ仮引数では、呼び出し時にそれに対応する **実引数** は省略でき、その場合は仮引数のデフォルト値が使われます。ある引数がデフォルト値を持っている場合、それ以降 "*" が出てくるまでの引数は全てデフォルト値を持っていなければなりません -- これは文法定義では表現されていない構文的制限です。

デフォルト引数値は関数定義が実行されるときに左から右へ評価されます。これは、デフォルト引数の式は関数が定義されるときにただ一度だけ評価され、同じ " 計算済みの" 値が呼び出しのたびに使用されることを意味します。この仕様を理解しておくことは特に、デフォルト引数値がリストや辞書のようなミュータブルなオブジェクトであるときに重要です: 関数がこのオブジェクトを変更 (例えばリストに要素を追加) すると、このデフォルト引数値が変更の影響を受けてしまします。一般には、これは意図しない動作です。このような動作を避けるには、デフォルト値として `None` を使い、この値を関数本体の中で明示的にテストします。例えば以下のようにします:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

関数呼び出しの意味付けに関する詳細は、**呼び出し** *(call)* 節で述べられています。関数呼び出しを行うと、パラメタリストに記述された全てのパラメタに、位置引数、キーワード引数、デフォルト値のいずれかから値が代入されます。"`*identifier`" 形式が存在すれば、余ったすべての位置引数を受け取ったタプルに初期化されます。このデフォルト値は空のタプルです。"`**identifier`" 形式が存在すれば、余ったすべてのキーワード引数を受け取った順序付きのマッピングオブジェクトに初期化されます。このデフォルト値は同じ型の空のマッピングオブジェクトです。"*" や "`*identifier`" の後のパラメタはキーワード専用パラメータで、キーワード引数によってのみ渡されます。"/" の前のパラメタは位置専用パラメータで、位置引数によってのみ渡されます。

バージョン 3.8 で変更: The / function parameter syntax may be used to indicate positional-only parameters. See **PEP 570** for details.

引数には、引数名に続けて "`: expression`" 形式の **アノテーション** を付けられます。`*identifier` や `**identifier` の形式でも、すべての引数にはアノテーションをつけられます。関数には、引数リストの後に "`-> expression`" 形式の "return" アノテーションをつけられます。これらのアノテーションは、任意の有効な Python の式が使えます。アノテーションがあっても、関数の意味論は変わりません。アノテーションの値は、関数オブジェクトの `__annotations__` 属性の、引数名をキーとする値として得られます。`__future__` の `annotations` インポートを使った場合は、アノテーションは実行時には文字列として保持され、これにより評価

の遅延が可能になっています。そうでない場合は、アノテーションは関数定義が実行されたときに評価されます。このケースでは、アノテーションはソースコードに現れたのとは違う順序で評価されることがあります。

式を即時に使用するために、無名関数 (名前に束縛されていない関数) を作成することもできます。これは **ラムダ** *(lambda)* の節で解説されているラムダ式を使います。ラムダ式は簡略化された関数定義の簡略表現に過ぎないことに注意してください; "*def*" 文で定義された関数もラムダ式で作成された関数のように、引数として渡せたり、他の名前に割り当てることができます。複数の式とアノテーションが実行できるので、"def" 形式の方がより強力です。

プログラマへのメモ: 関数は第一級オブジェクトです。関数定義内で実行された "def" 文は、返り値や引数として渡せるローカル関数を定義します。ネストした関数内で使われる自由変数は、def を含んでいる関数のローカル変数にアクセスできます。詳細は **名前づけと束縛** *(naming and binding)* 節を参照してください。

参考

PEP 3107 - Function Annotations 関数アノテーションの元の仕様書。

PEP 484 - **型ヒント** アノテーションの標準的な意味付けである型ヒントの定義。

PEP 526 - Syntax for Variable Annotations
Ability to type hint variable declarations, including class variables and instance variables.

PEP 563 - **アノテーションの遅延評価** 実行時にアノテーションを貪欲評価するのではなく文字列形式で保持することによる、アノテーションにおける前方参照のサポート

PEP 318 - Decorators for Functions and Methods
Function and method decorators were introduced. Class decorators were introduced in **PEP 3129**.

8.8 クラス定義

クラス定義は、クラスオブジェクトを定義します (**標準型の階層** 節参照):

```
classdef      ::=  [decorators] "class" classname [type_params] [inheritance] ":" suite
inheritance   ::=  "(" [argument_list] ")"
classname     ::=  identifier
```

クラス定義は実行可能な文です。継承リストは通常、基底クラスリストを与えます (より高度な使い方は、**メタク**

ラス を参照してください)。ですから、リストのそれぞれの要素の評価はサブクラス化しても良いクラスであるべきです。継承リストのないクラスは、デフォルトで、基底クラス `object` を継承するので:

```
class Foo:
    pass
```

は、以下と同等です

```
class Foo(object):
    pass
```

次にクラスのスイートが、新たな実行フレーム (**名前づけと束縛** *(naming and binding)* を参照してください) 内で、新たに作られたローカル名前空間と元々のグローバル名前空間を使って実行されます (通常、このスイートには主に関数定義が含まれます)。クラスのスイートが実行し終えると、実行フレームは破棄されますが、ローカルな名前空間は保存されます。[*5] 次に、継承リストを基底クラスに、保存されたローカル名前空間を属性値辞書に、それぞれ使ってクラスオブジェクトが生成されます。最後に、もとのローカル名前空間において、クラス名がこのクラスオブジェクトに束縛されます。

クラス本体で属性が定義された順序は新しいクラスの `__dict__` に保持されます。この性質が期待できるのは、クラスが作られた直後かつ定義構文を使って定義されたクラスであるときのみです。

クラス作成は、**メタクラス** を利用して大幅にカスタマイズできます。

関数をデコレートするのと同じように、クラスもデコレートすることが出来ます、

```
@f1(arg)
@f2
class Foo: pass
```

は、だいたい次と等価です

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

デコレータ式の評価規則は関数デコレータと同じです。結果はクラス名に束縛されます。

バージョン 3.9 で変更: Classes may be decorated with any valid *`assignment_expression`*. Previously, the grammar was much more restrictive; see **PEP 614** for details.

A list of *type parameters* may be given in square brackets immediately after the class's name. This indicates to static type checkers that the class is generic. At runtime, the type parameters can be retrieved from the class's `__type_params__` attribute. See *Generic classes* for more.

バージョン 3.12 で変更: Type parameter lists are new in Python 3.12.

[*5] クラスの本体の最初の文として現われる文字列リテラルは、その名前空間の `__doc__` 要素となり、そのクラスの ドキュメンテーション文字列 になります。

プログラマのための注釈: クラス定義内で定義された変数はクラス属性であり、全てのインスタンス間で共有されます。インスタンス属性は、メソッドの中で `self.name = value` とすることで設定できます。クラス属性もインスタンス属性も "`self.name`" 表記でアクセスでき、この表記でアクセスしたとき、インスタンス属性は同名のクラス属性を隠蔽します。クラス属性は、インスタンス属性のデフォルト値として使えますが、そこにミュータブルな値を使うと予期せぬ結果につながります。記述子 を使うと、詳細な実装が異なるインスタンス変数を作成できます。

参考

PEP 3115 - Metaclasses in Python 3000
メタクラスの宣言を現在の文法と、メタクラス付きのクラスがどのように構築されるかの意味論を変更した提案

PEP 3129 - **クラスデコレータ**
クラスデコレータを追加した提案。関数デコレータとメソッドデコレータは **PEP 318** で導入されました。

8.9 コルーチン

Added in version 3.5.

8.9.1 コルーチン関数定義

```
async_funcdef   ::=   [decorators] "async" "def" funcname "(" [parameter_list] ")"
                      ["->" expression] ":" suite
```

Python で実行しているコルーチンは多くの時点で一時停止と再開ができます (*coroutine* を参照)。*await* 式である *async for* と *async with* はコルーチン関数の本体でしか使えません。

Functions defined with `async def` syntax are always coroutine functions, even if they do not contain `await` or `async` keywords.

コルーチン関数の本体の中で `yield from` 式を使用すると `SyntaxError` になります。

コルーチン関数の例:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

バージョン 3.7 で変更: `await` and `async` are now keywords; previously they were only treated as such inside the body of a coroutine function.

8.9.2 async for 文

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

asynchronous iterable は、その `__anext__` メソッドで非同期なコードを実行可能な、*asynchronous iterator* を直接返す `__aiter__` メソッドを提供しています。

`async for` 文によって非同期なイテラブルを簡単にイテレーションすることができます。

以下のコード:

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

は意味論的に以下と等価です:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True

while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        SUITE
else:
    SUITE2
```

詳細は *__aiter__()* や *__anext__()* を参照してください。

コルーチン関数の本体の外で `async for` 文を使用すると `SyntaxError` になります。

8.9.3 async with 文

```
async_with_stmt ::= "async" with_stmt
```

asynchronous context manager は、*enter* メソッドと *exit* メソッド内部で実行を一時停止できる *context manager* です。

以下のコード:

```
async with EXPRESSION as TARGET:
    SUITE
```

これは次と等価です:

```
manager = (EXPRESSION)
aenter = type(manager).__aenter__
aexit = type(manager).__aexit__
value = await aenter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit(manager, None, None, None)
```

詳細は *__aenter__()* や *__aexit__()* を参照してください。

コルーチン関数の本体の外で async with 文を使用すると SyntaxError になります。

参考

PEP 492 - async 構文および await 構文付きのコルーチン
コルーチンを Python のまともな独り立ちした概念にし、サポートする構文を追加した提案。

8.10 Type parameter lists

Added in version 3.12.

バージョン 3.13 で変更: Support for default values was added (see **PEP 696**).

```
type_params   ::=  "[" type_param ("," type_param)* "]"
type_param    ::=  typevar | typevartuple | paramspec
typevar       ::=  identifier (":" expression)? ("=" expression)?
typevartuple  ::=  "*" identifier ("=" expression)?
paramspec     ::=  "**" identifier ("=" expression)?
```

Functions (including *coroutines*), *classes* and *type aliases* may contain a type parameter list:

```
def max[T](args: list[T]) -> T:
    ...

async def amax[T](args: list[T]) -> T:
    ...

class Bag[T]:
    def __iter__(self) -> Iterator[T]:
        ...

    def add(self, arg: T) -> None:
        ...

type ListOrSet[T] = list[T] | set[T]
```

Semantically, this indicates that the function, class, or type alias is generic over a type variable. This information is primarily used by static type checkers, and at runtime, generic objects behave much like their non-generic counterparts.

Type parameters are declared in square brackets (`[]`) immediately after the name of the function, class, or type alias. The type parameters are accessible within the scope of the generic object, but not elsewhere. Thus, after a declaration `def func[T](): pass`, the name `T` is not available in the module scope. Below, the semantics of generic objects are described with more precision. The scope of type parameters is modeled with a special function (technically, an *annotation scope*) that wraps the creation of the generic object.

Generic functions, classes, and type aliases have a `__type_params__` attribute listing their type parameters.

Type parameters come in three kinds:

- `typing.TypeVar`, introduced by a plain name (e.g., `T`). Semantically, this represents a single type to a type checker.

- `typing.TypeVarTuple`, introduced by a name prefixed with a single asterisk (e.g., `*Ts`). Semantically, this stands for a tuple of any number of types.
- `typing.ParamSpec`, introduced by a name prefixed with two asterisks (e.g., `**P`). Semantically, this stands for the parameters of a callable.

`typing.TypeVar` declarations can define *bounds* and *constraints* with a colon (`:`) followed by an expression. A single expression after the colon indicates a bound (e.g. `T: int`). Semantically, this means that the `typing.TypeVar` can only represent types that are a subtype of this bound. A parenthesized tuple of expressions after the colon indicates a set of constraints (e.g. `T: (str, bytes)`). Each member of the tuple should be a type (again, this is not enforced at runtime). Constrained type variables can only take on one of the types in the list of constraints.

For `typing.TypeVar`s declared using the type parameter list syntax, the bound and constraints are not evaluated when the generic object is created, but only when the value is explicitly accessed through the attributes `__bound__` and `__constraints__`. To accomplish this, the bounds or constraints are evaluated in a separate *annotation scope*.

`typing.TypeVarTuple`s and `typing.ParamSpec`s cannot have bounds or constraints.

All three flavors of type parameters can also have a *default value*, which is used when the type parameter is not explicitly provided. This is added by appending a single equals sign (`=`) followed by an expression. Like the bounds and constraints of type variables, the default value is not evaluated when the object is created, but only when the type parameter's `__default__` attribute is accessed. To this end, the default value is evaluated in a separate *annotation scope*. If no default value is specified for a type parameter, the `__default__` attribute is set to the special sentinel object `typing.NoDefault`.

The following example indicates the full set of allowed type parameter declarations:

```
def overly_generic[
   SimpleTypeVar,
   TypeVarWithDefault = int,
   TypeVarWithBound: int,
   TypeVarWithConstraints: (str, bytes),
   *SimpleTypeVarTuple = (int, float),
   **SimpleParamSpec = (str, bytearray),
](
   a: SimpleTypeVar,
   b: TypeVarWithDefault,
   c: TypeVarWithBound,
   d: Callable[SimpleParamSpec, TypeVarWithConstraints],
   *e: SimpleTypeVarTuple,
): ...
```

8.10.1 Generic functions

Generic functions are declared as follows:

```
def func[T](arg: T): ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_func():
    T = typing.TypeVar("T")
    def func(arg: T): ...
    func.__type_params__ = (T,)
    return func
func = TYPE_PARAMS_OF_func()
```

Here `annotation-def` indicates an *annotation scope*, which is not actually bound to any name at runtime. (One other liberty is taken in the translation: the syntax does not go through attribute access on the `typing` module, but creates an instance of `typing.TypeVar` directly.)

The annotations of generic functions are evaluated within the annotation scope used for declaring the type parameters, but the function's defaults and decorators are not.

The following example illustrates the scoping rules for these cases, as well as for additional flavors of type parameters:

```
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = some_default):
    ...
```

Except for the *lazy evaluation* of the `TypeVar` bound, this is equivalent to:

```
DEFAULT_OF_arg = some_default

annotation-def TYPE_PARAMS_OF_func():

    annotation-def BOUND_OF_T():
        return int
    # In reality, BOUND_OF_T() is evaluated only on demand.
    T = typing.TypeVar("T", bound=BOUND_OF_T())

    Ts = typing.TypeVarTuple("Ts")
    P = typing.ParamSpec("P")

    def func(*args: *Ts, arg: Callable[P, T] = DEFAULT_OF_arg):
        ...

    func.__type_params__ = (T, Ts, P)
```

(次のページに続く)

(前のページからの続き)

```
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

The capitalized names like `DEFAULT_OF_arg` are not actually bound at runtime.

8.10.2 Generic classes

Generic classes are declared as follows:

```
class Bag[T]: ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(typing.Generic[T]):
        __type_params__ = (T,)
        ...
    return Bag
Bag = TYPE_PARAMS_OF_Bag()
```

Here again `annotation-def` (not a real keyword) indicates an *annotation scope*, and the name `TYPE_PARAMS_OF_Bag` is not actually bound at runtime.

Generic classes implicitly inherit from `typing.Generic`. The base classes and keyword arguments of generic classes are evaluated within the type scope for the type parameters, and decorators are evaluated outside that scope. This is illustrated by this example:

```
@decorator
class Bag(Base[T], arg=T): ...
```

これは次と等価です:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(Base[T], typing.Generic[T], arg=T):
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

8.10.3 Generic type aliases

The *type* statement can also be used to create a generic type alias:

```
type ListOrSet[T] = list[T] | set[T]
```

Except for the *lazy evaluation* of the value, this is equivalent to:

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")

    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
    # In reality, the value is lazily evaluated
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_params=(T,))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

Here, `annotation-def` (not a real keyword) indicates an *annotation scope*. The capitalized names like `TYPE_PARAMS_OF_ListOrSet` are not actually bound at runtime.

脚注

第

NINE

トップレベル要素

Python インタプリタは、標準入力や、プログラムの引数として与えられたスクリプト、対話的にタイプ入力された命令、モジュールのソースファイルなど、様々な入力源から入力を得ることができます。この章では、それぞれの場合に用いられる構文法について説明しています。

9.1 完全な Python プログラム

言語仕様の中では、その言語を処理するインタプリタがどのように起動されるかまで規定する必要はないのですが、完全な Python プログラムの概念を知っておくと役に立ちます。完全な Python プログラムは、最小限に初期化された環境: 全ての組み込み変数と標準モジュールが利用可能で、かつ `sys` (様々なシステムサービス)、`builtins` (組み込み関数、例外、および `None`)、`__main__` の 3 つを除く全てのモジュールが初期化されていない状態で動作します。`__main__` は、完全なプログラムを実行する際に、ローカルおよびグローバルな名前空間を提供するために用いられます。

完全な Python プログラムの構文は、下の節で述べるファイル入力のためのものです。

インタプリタは、対話的モード (interactive mode) で起動されることもあります; この場合、インタプリタは完全なプログラムを読んで実行するのではなく、一度に単一の実行文 (複合文のときもあります) を読み込んで実行します。初期状態の環境は、完全なプログラムを実行するときの環境と同じです; 各実行文は、`__main__` の名前空間内で実行されます。

完全なプログラムは 3 つの形式でインタープリタに渡せます: `-c` *string* コマンドラインオプションで、コマンドラインの第 1 引数で渡されるファイル、あるいは標準入力として渡します。ファイルや標準入力が tty デバイスだった場合、インタープリタは対話モードに入ります。それ以外の場合は、ファイルを完全なプログラムとして実行します。

9.2 ファイル入力

非対話的なファイルから読み出された入力は、全て同じ形式:

```
file_input   ::=   (NEWLINE | statement)*
```

をとります。この構文法は、以下の状況で用いられます:

- (ファイルや文字列内の) 完全な Python プログラムを構文解析するとき;
- モジュールを構文解析するとき;
- `exec()` で渡された文字列を構文解析するとき;

9.3 対話的入力

対話モードでの入力は、以下の文法の下に構文解析されます:

```
interactive_input   ::=   [stmt_list] NEWLINE | compound_stmt NEWLINE
```

対話モードでは、(トップレベルの) 複合文の最後に空白行を入れなくてはならないことに注意してください; これは、複合文の終端をパーザが検出するための手がかりとして必要です。

9.4 式入力

式入力には `eval()` が使われます。これは先頭の空白を無視します。`eval()` に対する文字列引数は、以下の形式をとらなければなりません:

```
eval_input   ::=   expression_list NEWLINE*
```

第

TEN

完全な文法仕様

以下に示すのは Python の完全な文法です。CPython のパーサを生成するための文法定義から直接導かれたものです (Grammar/python.gram を参照のこと)。ここに示すのは、コード生成およびエラー処理に関係する部分を省略したバージョンです。

文法の記法は EBNF と PEG との混合です。特に、`&` にシンボル、トークンまたは括弧でくくられたグループが続く場合、それは正の先読み (positive lookahead) (すなわち、文法にマッチするためには必要だが、マッチ後に消費されない) です。一方で、`!` は負の先読み (negative lookahead) (すなわち、マッチ **しない** ために必要) です。PEG の順序付き選択 (ordered choice) の区切り文字として (伝統的に使われている `/` ではなく) `|` を使用します。文法の記法について、詳しくは **PEP 617** を見てください。

```
# PEG grammar for Python



# ========================= START OF THE GRAMMAR =========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic syntax error, the
#       location of the generic failure of the first pass will be used (this avoids
#       reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#       (like any rule in PEG).
#
# Grammar Syntax (see PEP 617 for more information):
#
# rule_name: expression
```

(次のページに続く)

(前のページからの続き)

```
#   Optionally, a type can be included right after the rule name, which
#   specifies the return type of the C or Python function corresponding to the
#   rule:
# rule_name[return_type]: expression
#   If the return type is omitted, then a void * is returned in C and an Any in
#   Python.
# e1 e2
#   Match e1, then match e2.
# e1 | e2
#   Match e1 or e2.
#   The first alternative can also appear on the line after the rule name for
#   formatting purposes. In that case, a | must be used before the first
#   alternative, like so:
#       rule_name[return_type]:
#            | first_alt
#            | second_alt
# ( e )
#   Match e (allows also to use other operators in the group like '(e)*')
# [ e ] or e?
#   Optionally match e.
# e*
#   Match zero or more occurrences of e.
# e+
#   Match one or more occurrences of e.
# s.e+
#   Match one or more occurrences of e, separated by s. The generated parse tree
#   does not include the separator. This is otherwise identical to (e (s e)*).
# &e
#   Succeed if e can be parsed, without consuming any input.
# !e
#   Fail if e can be parsed, without consuming any input.
# ~
#   Commit to the current alternative, even if it fails to parse.
# &&e
#   Eager parse e. The parser will not backtrack and will immediately
#   fail with SyntaxError if e cannot be parsed.
#

# STARTING RULES
# ==============

file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER

# GENERAL STATEMENTS
```

(次のページに続く)

(前のページからの続き)

```
# ==================

statements: statement+

statement: compound_stmt  | simple_stmts

statement_newline:
    | compound_stmt NEWLINE
    | simple_stmts
    | NEWLINE
    | ENDMARKER

simple_stmts:
    | simple_stmt !';' NEWLINE  # Not needed, there for speedup
    | ';'.simple_stmt+ [';'] NEWLINE

# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt:
    | assignment
    | type_alias
    | star_expressions
    | return_stmt
    | import_stmt
    | raise_stmt
    | 'pass'
    | del_stmt
    | yield_stmt
    | assert_stmt
    | 'break'
    | 'continue'
    | global_stmt
    | nonlocal_stmt

compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | with_stmt
    | for_stmt
    | try_stmt
    | while_stmt
    | match_stmt

# SIMPLE STATEMENTS
# =================
```

(次のページに続く)

(前のページからの続き)

```
# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with 'yield'
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | ('(' single_target ')'
         | single_subscript_attribute_target) ':' expression ['=' annotated_rhs ]
    | (star_targets '=' )+ (yield_expr | star_expressions) !'=' [TYPE_COMMENT]
    | single_target augassign ~ (yield_expr | star_expressions)

annotated_rhs: yield_expr | star_expressions

augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

global_stmt: 'global' ','.NAME+

nonlocal_stmt: 'nonlocal' ','.NAME+

del_stmt:
    | 'del' del_targets &(';' | NEWLINE)

yield_stmt: yield_expr

assert_stmt: 'assert' expression [',' expression ]

import_stmt:
    | import_name
    | import_from
```

(次のページに続く)

(前のページからの続き)

```
# Import statements
# -----------------

import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from:
    | 'from' ('.' | '...')* dotted_name 'import' import_from_targets
    | 'from' ('.' | '...')+ 'import' import_from_targets
import_from_targets:
    | '(' import_from_as_names [','] ')'
    | import_from_as_names !','
    | '*'
import_from_as_names:
    | ','.import_from_as_name+
import_from_as_name:
    | NAME ['as' NAME ]
dotted_as_names:
    | ','.dotted_as_name+
dotted_as_name:
    | dotted_name ['as' NAME ]
dotted_name:
    | dotted_name '.' NAME
    | NAME

# COMPOUND STATEMENTS
# ===================

# Common elements
# ---------------

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmts

decorators: ('@' named_expression NEWLINE )+

# Class definitions
# -----------------

class_def:
    | decorators class_def_raw
    | class_def_raw

class_def_raw:
    | 'class' NAME [type_params] ['(' [arguments] ')' ] ':' block

# Function definitions
```

(次のページに続く)

(前のページからの続き)

```
# --------------------

function_def:
    | decorators function_def_raw
    | function_def_raw

function_def_raw:
    | 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_type_comment] block
    | 'async' 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_type_comment]␣
↪block

# Function parameters
# --------------------

params:
    | parameters

parameters:
    | slash_no_default param_no_default* param_with_default* [star_etc]
    | slash_with_default param_with_default* [star_etc]
    | param_no_default+ param_with_default* [star_etc]
    | param_with_default+ [star_etc]
    | star_etc

# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).

slash_no_default:
    | param_no_default+ '/' ','
    | param_no_default+ '/' &')'
slash_with_default:
    | param_no_default* param_with_default+ '/' ','
    | param_no_default* param_with_default+ '/' &')'

star_etc:
    | '*' param_no_default param_maybe_default* [kwds]
    | '*' param_no_default_star_annotation param_maybe_default* [kwds]
    | '*' ',' param_maybe_default+ [kwds]
    | kwds

kwds:
    | '**' param_no_default

# One parameter.  This *includes* a following comma and type comment.
#
# There are three styles:
# - No default
```

(次のページに続く)

(前のページからの続き)

```
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
#

param_no_default:
    | param ',' TYPE_COMMENT?
    | param TYPE_COMMENT? &')'
param_no_default_star_annotation:
    | param_star_annotation ',' TYPE_COMMENT?
    | param_star_annotation TYPE_COMMENT? &')'
param_with_default:
    | param default ',' TYPE_COMMENT?
    | param default TYPE_COMMENT? &')'
param_maybe_default:
    | param default? ',' TYPE_COMMENT?
    | param default? TYPE_COMMENT? &')'
param: NAME annotation?
param_star_annotation: NAME star_annotation
annotation: ':' expression
star_annotation: ':' star_expression
default: '=' expression  | invalid_default

# If statement
# ------------

if_stmt:
    | 'if' named_expression ':' block elif_stmt
    | 'if' named_expression ':' block [else_block]
elif_stmt:
    | 'elif' named_expression ':' block elif_stmt
    | 'elif' named_expression ':' block [else_block]
else_block:
    | 'else' ':' block

# While statement
# ---------------

while_stmt:
    | 'while' named_expression ':' block [else_block]

# For statement
# -------------
```

(次のページに続く)

(前のページからの続き)

```
for_stmt:
    | 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_block]
    | 'async' 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_block]

# With statement
# --------------

with_stmt:
    | 'with' '(' ','.with_item+ ','? ')' ':' [TYPE_COMMENT] block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | 'async' 'with' '(' ','.with_item+ ','? ')' ':' block
    | 'async' 'with' ','.with_item+ ':' [TYPE_COMMENT] block

with_item:
    | expression 'as' star_target &(',' | ')' | ':')
    | expression

# Try statement
# -------------

try_stmt:
    | 'try' ':' block finally_block
    | 'try' ':' block except_block+ [else_block] [finally_block]
    | 'try' ':' block except_star_block+ [else_block] [finally_block]


# Except statement
# ----------------

except_block:
    | 'except' expression ['as' NAME ] ':' block
    | 'except' ':' block
except_star_block:
    | 'except' '*' expression ['as' NAME ] ':' block
finally_block:
    | 'finally' ':' block

# Match statement
# ---------------

match_stmt:
    | "match" subject_expr ':' NEWLINE INDENT case_block+ DEDENT

subject_expr:
    | star_named_expression ',' star_named_expressions?
    | named_expression
```

(次のページに続く)

(前のページからの続き)

```
case_block:
    | "case" patterns guard? ':' block

guard: 'if' named_expression

patterns:
    | open_sequence_pattern
    | pattern

pattern:
    | as_pattern
    | or_pattern

as_pattern:
    | or_pattern 'as' pattern_capture_target

or_pattern:
    | '|'.closed_pattern+

closed_pattern:
    | literal_pattern
    | capture_pattern
    | wildcard_pattern
    | value_pattern
    | group_pattern
    | sequence_pattern
    | mapping_pattern
    | class_pattern

# Literal patterns are used for equality and identity constraints
literal_pattern:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

# Literal expressions are used to restrict permitted mapping pattern keys
literal_expr:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'
```

(次のページに続く)

(前のページからの続き)

```
complex_number:
    | signed_real_number '+' imaginary_number
    | signed_real_number '-' imaginary_number

signed_number:
    | NUMBER
    | '-' NUMBER

signed_real_number:
    | real_number
    | '-' real_number

real_number:
    | NUMBER

imaginary_number:
    | NUMBER

capture_pattern:
    | pattern_capture_target

pattern_capture_target:
    | !"_" NAME !('.' | '(' | '=')

wildcard_pattern:
    | "_"

value_pattern:
    | attr !('.' | '(' | '=')

attr:
    | name_or_attr '.' NAME

name_or_attr:
    | attr
    | NAME

group_pattern:
    | '(' pattern ')'

sequence_pattern:
    | '[' maybe_sequence_pattern? ']'
    | '(' open_sequence_pattern? ')'

open_sequence_pattern:
    | maybe_star_pattern ',' maybe_sequence_pattern?
```

(次のページに続く)

(前のページからの続き)

```
maybe_sequence_pattern:
    | ','.maybe_star_pattern+ ','?

maybe_star_pattern:
    | star_pattern
    | pattern

star_pattern:
    | '*' pattern_capture_target
    | '*' wildcard_pattern

mapping_pattern:
    | '{' '}'
    | '{' double_star_pattern ','? '}'
    | '{' items_pattern ',' double_star_pattern ','? '}'
    | '{' items_pattern ','? '}'

items_pattern:
    | ','.key_value_pattern+

key_value_pattern:
    | (literal_expr | attr) ':' pattern

double_star_pattern:
    | '**' pattern_capture_target

class_pattern:
    | name_or_attr '(' ')'
    | name_or_attr '(' positional_patterns ','? ')'
    | name_or_attr '(' keyword_patterns ','? ')'
    | name_or_attr '(' positional_patterns ',' keyword_patterns ','? ')'

positional_patterns:
    | ','.pattern+

keyword_patterns:
    | ','.keyword_pattern+

keyword_pattern:
    | NAME '=' pattern

# Type statement
# ---------------

type_alias:
    | "type" NAME [type_params] '=' expression
```

(次のページに続く)

(前のページからの続き)

```
# Type parameter declaration
# --------------------------

type_params:
    | invalid_type_params
    | '[' type_param_seq ']'

type_param_seq: ','.type_param+ [',']

type_param:
    | NAME [type_param_bound] [type_param_default]
    | '*' NAME [type_param_starred_default]
    | '**' NAME [type_param_default]

type_param_bound: ':' expression
type_param_default: '=' expression
type_param_starred_default: '=' star_expression

# EXPRESSIONS
# -----------

expressions:
    | expression (',' expression )+ [',']
    | expression ','
    | expression

expression:
    | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef

yield_expr:
    | 'yield' 'from' expression
    | 'yield' [star_expressions]

star_expressions:
    | star_expression (',' star_expression )+ [',']
    | star_expression ','
    | star_expression

star_expression:
    | '*' bitwise_or
    | expression

star_named_expressions: ','.star_named_expression+ [',']
```

(次のページに続く)

(前のページからの続き)

```
star_named_expression:
    | '*' bitwise_or
    | named_expression

assignment_expression:
    | NAME ':=' ~ expression

named_expression:
    | assignment_expression
    | expression !':='

disjunction:
    | conjunction ('or' conjunction )+
    | conjunction

conjunction:
    | inversion ('and' inversion )+
    | inversion

inversion:
    | 'not' inversion
    | comparison

# Comparison operators
# --------------------

comparison:
    | bitwise_or compare_op_bitwise_or_pair+
    | bitwise_or

compare_op_bitwise_or_pair:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
    | notin_bitwise_or
    | in_bitwise_or
    | isnot_bitwise_or
    | is_bitwise_or

eq_bitwise_or: '==' bitwise_or
noteq_bitwise_or:
    | ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or
lt_bitwise_or: '<' bitwise_or
```

(次のページに続く)

(前のページからの続き)

```
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or

# Bitwise operators
# -----------------

bitwise_or:
    | bitwise_or '|' bitwise_xor
    | bitwise_xor

bitwise_xor:
    | bitwise_xor '^' bitwise_and
    | bitwise_and

bitwise_and:
    | bitwise_and '&' shift_expr
    | shift_expr

shift_expr:
    | shift_expr '<<' sum
    | shift_expr '>>' sum
    | sum

# Arithmetic operators
# --------------------

sum:
    | sum '+' term
    | sum '-' term
    | term

term:
    | term '*' factor
    | term '/' factor
    | term '//' factor
    | term '%' factor
    | term '@' factor
    | factor

factor:
    | '+' factor
    | '-' factor
    | '~' factor
```

(次のページに続く)

(前のページからの続き)

```
    | power

power:
    | await_primary '**' factor
    | await_primary

# Primary elements
# ----------------

# Primary elements are things like "obj.something.something", "obj[something]", "obj(something)",
↪"obj" ...

await_primary:
    | 'await' primary
    | primary

primary:
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice !','
    | ','.(slice | starred_expression)+ [',']

slice:
    | [expression] ':' [expression] [':' [expression] ]
    | named_expression

atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

group:
    | '(' (yield_expr | named_expression) ')'

# Lambda functions
```

(次のページに続く)

(前のページからの続き)

```
# ----------------

lambdef:
    | 'lambda' [lambda_params] ':' expression

lambda_params:
    | lambda_parameters

# lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis.  (For more, see parameters above.)
#
lambda_parameters:
    | lambda_slash_no_default lambda_param_no_default* lambda_param_with_default* [lambda_star_etc]
    | lambda_slash_with_default lambda_param_with_default* [lambda_star_etc]
    | lambda_param_no_default+ lambda_param_with_default* [lambda_star_etc]
    | lambda_param_with_default+ [lambda_star_etc]
    | lambda_star_etc

lambda_slash_no_default:
    | lambda_param_no_default+ '/' ','
    | lambda_param_no_default+ '/' &':'

lambda_slash_with_default:
    | lambda_param_no_default* lambda_param_with_default+ '/' ','
    | lambda_param_no_default* lambda_param_with_default+ '/' &':'

lambda_star_etc:
    | '*' lambda_param_no_default lambda_param_maybe_default* [lambda_kwds]
    | '*' ',' lambda_param_maybe_default+ [lambda_kwds]
    | lambda_kwds

lambda_kwds:
    | '**' lambda_param_no_default

lambda_param_no_default:
    | lambda_param ','
    | lambda_param &':'
lambda_param_with_default:
    | lambda_param default ','
    | lambda_param default &':'
lambda_param_maybe_default:
    | lambda_param default? ','
    | lambda_param default? &':'
lambda_param: NAME

# LITERALS
```

(次のページに続く)

(前のページからの続き)

```
# ========

fstring_middle:
    | fstring_replacement_field
    | FSTRING_MIDDLE
fstring_replacement_field:
    | '{' annotated_rhs '='? [fstring_conversion] [fstring_full_format_spec] '}'
fstring_conversion:
    | "!" NAME
fstring_full_format_spec:
    | ':' fstring_format_spec*
fstring_format_spec:
    | FSTRING_MIDDLE
    | fstring_replacement_field
fstring:
    | FSTRING_START fstring_middle* FSTRING_END

string: STRING
strings: (fstring|string)+

list:
    | '[' [star_named_expressions] ']'

tuple:
    | '(' [star_named_expression ',' [star_named_expressions]  ] ')'

set: '{' star_named_expressions '}'

# Dicts
# -----

dict:
    | '{' [double_starred_kvpairs] '}'

double_starred_kvpairs: ','.double_starred_kvpair+ [',']

double_starred_kvpair:
    | '**' bitwise_or
    | kvpair

kvpair: expression ':' expression

# Comprehensions & Generators
# ---------------------------

for_if_clauses:
    | for_if_clause+
```

(次のページに続く)

(前のページからの続き)

```
for_if_clause:
    | 'async' 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
    | 'for' star_targets 'in' ~ disjunction ('if' disjunction )*

listcomp:
    | '[' named_expression for_if_clauses ']'

setcomp:
    | '{' named_expression for_if_clauses '}'

genexp:
    | '(' ( assignment_expression | expression !':=') for_if_clauses ')'

dictcomp:
    | '{' kvpair for_if_clauses '}'

# FUNCTION CALL ARGUMENTS
# =======================

arguments:
    | args [','] &')'

args:
    | ','.(starred_expression | ( assignment_expression | expression !':=') !'=')+ [',' kwargs ]
    | kwargs

kwargs:
    | ','.kwarg_or_starred+ ',' ','.kwarg_or_double_starred+
    | ','.kwarg_or_starred+
    | ','.kwarg_or_double_starred+

starred_expression:
    | '*' expression

kwarg_or_starred:
    | NAME '=' expression
    | starred_expression

kwarg_or_double_starred:
    | NAME '=' expression
    | '**' expression

# ASSIGNMENT TARGETS
# ==================

# Generic targets
```

(次のページに続く)

(前のページからの続き)

```
# ---------------

# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
    | star_target !','
    | star_target (',' star_target )* [',']

star_targets_list_seq: ','.star_target+ [',']

star_targets_tuple_seq:
    | star_target (',' star_target )+ [',']
    | star_target ','

star_target:
    | '*' (!'*' star_target)
    | target_with_star_atom

target_with_star_atom:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom

star_atom:
    | NAME
    | '(' target_with_star_atom ')'
    | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'

single_target:
    | single_subscript_attribute_target
    | NAME
    | '(' single_target ')'

single_subscript_attribute_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead

t_primary:
    | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
    | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead
    | atom &t_lookahead

t_lookahead: '(' | '[' | '.'

# Targets for del statements
```

(次のページに続く)

(前のページからの続き)

```
# --------------------------

del_targets: ','.del_target+ [',']

del_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | del_t_atom

del_t_atom:
    | NAME
    | '(' del_target ')'
    | '(' [del_targets] ')'
    | '[' [del_targets] ']'

# TYPING ELEMENTS
# ---------------

# type_expressions allow */** but ignore them
type_expressions:
    | ','.expression+ ',' '*' expression ',' '**' expression
    | ','.expression+ ',' '*' expression
    | ','.expression+ ',' '**' expression
    | '*' expression ',' '**' expression
    | '*' expression
    | '**' expression
    | ','.expression+

func_type_comment:
    | NEWLINE TYPE_COMMENT &(NEWLINE INDENT)   # Must be followed by indented block
    | TYPE_COMMENT

# ========================= END OF THE GRAMMAR ===========================



# ========================= START OF INVALID RULES ========================
```

付録

A

用語集

>>> 対話型 (*interactive*) シェルにおけるデフォルトの Python プロンプトです。インタープリターで対話的に実行されるコード例でよく見られます。

... 次のものが考えられます:

- 対話型 (*interactive*) シェルにおいて、インデントされたコードブロック、対応する左右の区切り文字の組 (丸括弧、角括弧、波括弧、三重引用符) の内側、デコレーターの後に、コードを入力する際に表示されるデフォルトの Python プロンプトです。
- 組み込みの定数 `Ellipsis` 。

abstract base class
: (抽象基底クラス) 抽象基底クラスは *duck-typing* を補完するもので、`hasattr()` などの別のテクニックでは不恰好であったり微妙に誤る (例えば *magic methods* の場合) 場合にインターフェースを定義する方法を提供します。ABC は仮想 (virtual) サブクラスを導入します。これは親クラスから継承しませんが、それでも `isinstance()` や `issubclass()` に認識されます; `abc` モジュールのドキュメントを参照してください。Python には、多くの組み込み ABC が同梱されています。その対象は、(`collections.abc` モジュールで) データ構造、(`numbers` モジュールで) 数、(`io` モジュールで) ストリーム、(`importlib.abc` モジュールで) インポートファインダ及びローダーです。`abc` モジュールを利用して独自の ABC を作成できます。

annotation
: (アノテーション) 変数、クラス属性、関数のパラメータや返り値に関係するラベルです。慣例により *type hint* として使われています。

  ローカル変数のアノテーションは実行時にはアクセスできませんが、グローバル変数、クラス属性、関数のアノテーションはそれぞれモジュール、クラス、関数の `__annotations__` 特殊属性に保持されています。

  機能の説明がある *variable annotation*, *function annotation*, **PEP 484**, **PEP 526** を参照してください。また、アノテーションを利用するベストプラクティスとして annotations-howto も参照してください。

引数 (argument)

(実引数) 関数を呼び出す際に、**関数** (または **メソッド**) に渡す値です。実引数には 2 種類あります:

- **キーワード引数**: 関数呼び出しの際に引数の前に識別子がついたもの (例: `name=`) や、`**` に続けた辞書の中の値として渡された引数。例えば、次の `complex()` の呼び出しでは、`3` と `5` がキーワード引数です:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- **位置引数**: キーワード引数以外の引数。位置引数は引数リストの先頭に書くことができ、また `*` に続けた *iterable* の要素として渡すことができます。例えば、次の例では `3` と `5` は両方共位置引数です:

```
complex(3, 5)
complex(*(3, 5))
```

実引数は関数の実体において名前付きのローカル変数に割り当てられます。割り当てを行う規則については **呼び出し** *(call)* を参照してください。シンタックスにおいて実引数を表すためにあらゆる式を使うことが出来ます。評価された値はローカル変数に割り当てられます。

仮引数 、FAQ の 実引数と仮引数の違いは何ですか? 、**PEP 362** を参照してください。

asynchronous context manager

(非同期コンテキストマネージャ) *`__aenter__()`* と *`__aexit__()`* メソッドを定義することで *`async with`* 文内の環境を管理するオブジェクトです。**PEP 492** で導入されました。

asynchronous generator

(非同期ジェネレータ) *asynchronous generator iterator* を返す関数です。*`async def`* で定義されたコルーチン関数に似ていますが、*`yield`* 式を持つ点で異なります。*`yield`* 式は *`async for`* ループで使用できる値の並びを生成するのに使用されます。

通常は非同期ジェネレータ関数を指しますが、文脈によっては **非同期ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

非同期ジェネレータ関数には、*`async for`* 文や *`async with`* 文だけでなく *`await`* 式もあることがあります。

asynchronous generator iterator

(非同期ジェネレータイテレータ) *asynchronous generator* 関数で生成されるオブジェクトです。

これは *asynchronous iterator* で、*`__anext__()`* メソッドを使って呼ばれると awaitable オブジェクトを返します。この awaitable オブジェクトは、次の *`yield`* 式まで非同期ジェネレータ関数の本体を実行します。

各 *`yield`* では一時的に処理を中断し、その場の実行状態 (ローカル変数や保留中の try 文を含む) を記憶します。**非同期ジェネレータイテレータ** が *`__anext__()`* で返された他の awaitable で実際に再開する時

には、その中断箇所が選ばれます。**PEP 492** および **PEP 525** を参照してください。

asynchronous iterable

(非同期イテラブル) *async for* 文の中で使用できるオブジェクトです。自身の *__aiter__()* メソッドから *asynchronous iterator* を返さなければなりません。**PEP 492** で導入されました。

asynchronous iterator

(非同期イテレータ) *__aiter__()* と *__anext__()* メソッドを実装したオブジェクトです。*__anext__()* は *awaitable* オブジェクトを返さなければなりません。*async for* は `StopAsyncIteration` 例外を送出するまで、非同期イテレータの *__anext__()* メソッドが返す awaitable を解決します。**PEP 492** で導入されました。

属性

(属性) オブジェクトに関連付けられ、ドット表記式によって名前で通常参照される値です。例えば、オブジェクト *o* が属性 *a* を持っているとき、その属性は *o.a* で参照されます。

オブジェクトには、**識別子** *(identifier)* **およびキーワード** *(keyword)* で定義される識別子ではない名前の属性を与えることができます。たとえば `setattr()` を使い、オブジェクトがそれを許可している場合に行えます。このような属性はドット表記式ではアクセスできず、代わりに `getattr()` を使って取る必要があります。

awaitable

(待機可能) *await* 式で使用することが出来るオブジェクトです。*coroutine* か、*__await__()* メソッドがあるオブジェクトです。**PEP 492** を参照してください。

BDFL 慈

悲深き終身独裁者 (Benevolent Dictator For Life) の略です。Python の作者、Guido van Rossum のことです。

binary file

(バイナリファイル) *bytes-like* **オブジェクト** の読み込みおよび書き込みができる **ファイルオブジェクト** です。バイナリファイルの例は、バイナリモード (`'rb'`, `'wb'` or `'rb+'`) で開かれたファイル、`sys.stdin.buffer`、`sys.stdout.buffer`、`io.BytesIO` や `gzip.GzipFile` のインスタンスです。

`str` オブジェクトの読み書きができるファイルオブジェクトについては、*text file* も参照してください。

borrowed reference In

Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

bytes-like object

bufferobjects をサポートしていて、C 言語の意味で **連続した** バッファーを提供可能なオブジェクト。`bytes`, `bytearray`, `array.array` や、多くの一般的な `memoryview` オブジェクトがこれに当たります。bytes-like オブジェクトは、データ圧縮、バイナリファイルへの保存、ソケットを経由した送信など、バイナリデータを要求するいろいろな操作に利用することができます。

幾つかの操作ではバイナリデータを変更する必要があります。その操作のドキュメントではよく ” 読み書き可能な bytes-like オブジェクト” に言及しています。変更可能なバッファーオブジェクトには、`bytearray` と `bytearray` の `memoryview` などが含まれます。また、他の幾つかの操作では不変なオブジェクト内のバイナリデータ (” 読み出し専用の bytes-like オブジェクト”) を必要します。それには `bytes` と `bytes` の `memoryview` オブジェクトが含まれます。

bytecode

(バイトコード) Python のソースコードは、Python プログラムの CPython インタプリタの内部表現であるバイトコードへとコンパイルされます。バイトコードは `.pyc` ファイルにキャッシュされ、同じファイルが二度目に実行されるときはより高速になります (ソースコードからバイトコードへの再度のコンパイルは回避されます)。この ” 中間言語 (intermediate language)” は、各々のバイトコードに対応する機械語を実行する **仮想マシン** で動作するといえます。重要な注意として、バイトコードは異なる Python 仮想マシン間で動作することや、Python リリース間で安定であることは期待されていません。

バイトコードの命令一覧は dis モジュール にあります。

callable

A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the *__call__()* method is also a callable.

callback

(コールバック) 将来のある時点で実行されるために引数として渡される関数

クラス

(クラス) ユーザー定義オブジェクトを作成するためのテンプレートです。クラス定義は普通、そのクラスのインスタンス上の操作をするメソッドの定義を含みます。

class variable

(クラス変数) クラス上に定義され、クラスレベルで (つまり、クラスのインスタンス上ではなしに) 変更されることを目的としている変数です。

complex number

(複素数) よく知られている実数系を拡張したもので、すべての数は実部と虚部の和として表されます。虚数は虚数単位 (`-1` の平方根) に実数を掛けたもので、一般に数学では `i` と書かれ、工学では `j` と書かれます。Python は複素数に組み込みで対応し、後者の表記を取っています。虚部は末尾に `j` をつけて書きま

す。例えば 3+1j です。math モジュールの複素数版を利用するには、cmath を使います。複素数の使用はかなり高度な数学の機能です。必要性を感じなければ、ほぼ間違いなく無視してしまってよいでしょう。

context manager
: An object which controls the environment seen in a *with* statement by defining *__enter__()* and *__exit__()* methods. See **PEP 343**.

context variable
: (コンテキスト変数) コンテキストに依存して異なる値を持つ変数。これは、ある変数の値が各々の実行スレッドで異なり得るスレッドローカルストレージに似ています。しかしコンテキスト変数では、1 つの実行スレッドにいくつかのコンテキストがあり得、コンテキスト変数の主な用途は並列な非同期タスクの変数の追跡です。contextvars を参照してください。

contiguous
: (隣接、連続) バッファが厳密に *C*-**連続** または *Fortran* **連続** である場合に、そのバッファは連続しているとみなせます。ゼロ次元バッファは C 連続であり Fortran 連続です。一次元の配列では、その要素は必ずメモリ上で隣接するように配置され、添字がゼロから始まり増えていく順序で並びます。多次元の C-連続な配列では、メモリアドレス順に要素を巡る際には最後の添え字が最初に変わるのに対し、Fortran 連続な配列では最初の添え字が最初に動きます。

コルーチン
: (コルーチン) コルーチンはサブルーチンのより一般的な形式です。サブルーチンには決められた地点から入り、別の決められた地点から出ます。コルーチンには多くの様々な地点から入る、出る、再開することができます。コルーチンは *async def* 文で実装できます。**PEP 492** を参照してください。

coroutine function
: (コルーチン関数) *coroutine* オブジェクトを返す関数です。コルーチン関数は *async def* 文で実装され、*await*、*async for*、および *async with* キーワードを持つことが出来ます。これらは **PEP 492** で導入されました。

CPython
: python.org で配布されている、Python プログラミング言語の標準的な実装です。"CPython" という単語は、この実装を Jython や IronPython といった他の実装と区別する必要が有る場合に利用されます。

decorator
: (デコレータ) 別の関数を返す関数で、通常、@wrapper 構文で関数変換として適用されます。デコレータの一般的な利用例は、classmethod() と staticmethod() です。

  デコレータの文法はシンタックスシュガーです。次の 2 つの関数定義は意味的に同じものです:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
```

(次のページに続く)

(前のページからの続き)

```
def f(arg):
    ...
```

同じ概念がクラスにも存在しますが、あまり使われません。デコレータについて詳しくは、**関数定義** および **クラス定義** のドキュメントを参照してください。

descriptor

Any object which defines the methods *__get__()*, *__set__()*, or *__delete__()*. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

デスクリプタのメソッドに関しての詳細は、**デスクリプタ** *(descriptor)* **の実装** や Descriptor How To Guide を参照してください。

dictionary

An associative array, where arbitrary keys are mapped to values. The keys can be any object with *__hash__()* and *__eq__()* methods. Called a hash in Perl.

dictionary comprehension

(辞書内包表記) iterable 内の全てあるいは一部の要素を処理して、その結果からなる辞書を返すコンパクトな書き方です。`results = {n: n ** 2 for n in range(10)}` とすると、キー `n` を値 `n ** 2` に対応付ける辞書を生成します。**リスト、集合、辞書の表示** を参照してください。

dictionary view

(辞書ビュー) `dict.keys()`、`dict.values()`、`dict.items()` が返すオブジェクトです。辞書の項目の動的なビューを提供します。すなわち、辞書が変更されるとビューはそれを反映します。辞書ビューを強制的に完全なリストにするには `list(dictview)` を使用してください。dict-views を参照してください。

docstring

A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing

あるオブジェクトが正しいインターフェースを持っているかを決定するのにオブジェクトの型を見ないプログラミングスタイルです。代わりに、単純にオブジェクトのメソッドや属性が呼ばれたり使われたりします。(「アヒルのように見えて、アヒルのように鳴けば、それはアヒルである。」) インターフェースを型より重視することで、上手くデザインされたコードは、ポリモーフィックな代替を許して柔軟性を向上させます。ダックタイピングは `type()` や `isinstance()` による判定を避けます。(ただし、ダックタイピングを

抽象基底クラス で補完することもできます。) その代わり、典型的に `hasattr()` 判定や *EAFP* プログラミングを利用します。

EAFP
: 「認可をとるより許しを請う方が容易 (easier to ask for forgiveness than permission、マーフィーの法則)」の略です。この Python で広く使われているコーディングスタイルでは、通常は有効なキーや属性が存在するものと仮定し、その仮定が誤っていた場合に例外を捕捉します。この簡潔で手早く書けるコーディングスタイルには、*try* 文および *except* 文がたくさんあるのが特徴です。このテクニックは、C のような言語でよく使われている *LBYL* スタイルと対照的なものです。

expression
: (式) 何かの値と評価される、一まとまりの構文 (a piece of syntax) です。言い換えると、式とはリテラル、名前、属性アクセス、演算子や関数呼び出しなど、値を返す式の要素の積み重ねです。他の多くの言語と違い、Python では言語の全ての構成要素が式というわけではありません。*while* のように、式としては使えない 文 もあります。代入も式ではなく文です。

extension module
: (拡張モジュール) C や C++ で書かれたモジュールで、Python の C API を利用して Python コアやユーザーコードとやりとりします。

f-string
: `'f'` や `'F'` が先頭に付いた文字列リテラルは "f-string" と呼ばれ、これは フォーマット済み文字列リテラル の短縮形の名称です。**PEP 498** も参照してください。

file object
: An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

  ファイルオブジェクトには実際には 3 種類あります: 生の バイナリーファイル、バッファされた バイナリーファイル、そして テキストファイル です。インターフェイスは `io` モジュールで定義されています。ファイルオブジェクトを作る標準的な方法は `open()` 関数を使うことです。

file-like object
: *file object* と同義です。

filesystem encoding and error handler
: Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

  ファイルシステムのエンコーディングでは、すべてが 128 バイト以下に正常にデコードされることが保証されなくてはなりません。ファイルシステムのエンコーディングでこれが保証されなかった場合は、API 関数が `UnicodeError` を送出することがあります。

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

See also the *locale encoding*.

finder

(ファインダ) インポートされているモジュールの *loader* の発見を試行するオブジェクトです。

There are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See **インポートシステム** and `importlib` for much more detail.

floor division

(切り捨て除算) 一番近い整数に切り捨てる数学的除算。切り捨て除算演算子は `//` です。例えば、`11 // 4` は `2` になり、それとは対称に浮動小数点数の真の除算では `2.75` が 返ってきます。`(-11) // 4` は `-2.75` を **小さい方に** 丸める (訳注: 負の無限大への丸めを行う) ので `-3` になることに注意してください。**PEP 238** を参照してください。

free threading

A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

関数

(関数) 呼び出し側に値を返す一連の文のことです。関数には 0 以上の **実引数** を渡すことが出来ます。実体の実行時に引数を使用することが出来ます。**仮引数**、**メソッド**、**関数定義** を参照してください。

function annotation

(関数アノテーション) 関数のパラメータや返り値の *annotation* です。

関数アノテーションは、通常は **型ヒント** のために使われます: 例えば、この関数は 2 つの `int` 型の引数を取ると期待され、また `int` 型の返り値を持つと期待されています。

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

関数アノテーションの文法は **関数定義** の節で解説されています。

機能の説明がある *variable annotation*, **PEP 484**, を参照してください。また、アノテーションを利用するベストプラクティスとして annotations-howto も参照してください。

__future__

`from __future__ import <feature>` という *future* **文** は、コンパイラーに将来の Python リリース

で標準となる構文や意味を使用して現在のモジュールをコンパイルするよう指示します。`__future__` モジュールでは、*feature* のとりうる値をドキュメント化しています。このモジュールをインポートし、その変数を評価することで、新機能が最初に言語に追加されたのはいつかや、いつデフォルトになるか (またはなったか) を見ることができます:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection

(ガベージコレクション) これ以降使われることのないメモリを解放する処理です。Python は、参照カウントと、循環参照を検出し破壊する循環ガベージコレクタを使ってガベージコレクションを行います。ガベージコレクタは `gc` モジュールを使って操作できます。

ジェネレータ

(ジェネレータ) *generator iterator* を返す関数です。通常の関数に似ていますが、*`yield`* 式を持つ点で異なります。*`yield`* 式は、for ループで使用できたり、`next()` 関数で値を 1 つずつ取り出したりできる、値の並びを生成するのに使用されます。

通常はジェネレータ関数を指しますが、文脈によっては **ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

generator iterator

(ジェネレータイテレータ) *generator* 関数で生成されるオブジェクトです。

`yield` のたびに局所実行状態 (局所変数や未処理の try 文などを含む) を記憶して、処理は一時的に中断されます。**ジェネレータイテレータ** が再開されると、中断した位置を取得します (通常の関数が実行のたびに新しい状態から開始するのと対照的です)。

generator expression

An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function

(ジェネリック関数) 異なる型に対し同じ操作をする関数群から構成される関数です。呼び出し時にどの実装を用いるかはディスパッチアルゴリズムにより決定されます。

single dispatch、`functools.singledispatch()` デコレータ、**PEP 443** を参照してください。

generic type

A *type* that can be parameterized; typically a *container class* such as `list` or `dict`. Used for *type hints*

and *annotations*.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL

global interpreter lock を参照してください。

global interpreter lock

(グローバルインタプリタロック) *CPython* インタプリタが利用している、一度に Python の **バイトコード** を実行するスレッドは一つだけであることを保証する仕組みです。これにより (`dict` などの重要な組み込み型を含む) オブジェクトモデルが同時アクセスに対して暗黙的に安全になるので、CPython の実装がシンプルになります。インタプリタ全体をロックすることで、マルチプロセッサマシンが生じる並列化のコストと引き換えに、インタプリタを簡単にマルチスレッド化できるようになります。

ただし、標準あるいは外部のいくつかの拡張モジュールは、圧縮やハッシュ計算などの計算の重い処理をするときに GIL を解除するように設計されています。また、I/O 処理をする場合 GIL は常に解除されます。

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil 0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

hash-based pyc

(ハッシュベース pyc ファイル) 正当性を判別するために、対応するソースファイルの最終更新時刻ではなくハッシュ値を使用するバイトコードのキャッシュファイルです。**キャッシュされたバイトコードの無効化** を参照してください。

hashable

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a *`__hash__()`* method), and can be compared to other objects (it needs an *`__eq__()`* method). Hashable objects which compare equal must have the same hash value.

ハッシュ可能なオブジェクトは辞書のキーや集合のメンバーとして使えます。辞書や集合のデータ構造は内部でハッシュ値を使っているからです。

Python のイミュータブルな組み込みオブジェクトは、ほとんどがハッシュ可能です。(リストや辞書のような) ミュータブルなコンテナはハッシュ不可能です。(タプルや frozenset のような) イミュータブルなコンテナは、要素がハッシュ可能であるときのみハッシュ可能です。ユーザー定義のクラスのインスタンスであるようなオブジェクトはデフォルトでハッシュ可能です。それらは全て (自身を除いて) 比較結果は非等価であり、ハッシュ値は `id()` より得られます。

IDLE

Python の統合開発環境 (Integrated DeveLopment Environment) 及び学習環境 (Learning Environment) です。idle は Python の標準的な配布に同梱されている基本的な機能のエディタとインタプリタ環境です。

永続オブジェクト (immortal)

Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

immutable

(イミュータブル) 固定の値を持ったオブジェクトです。イミュータブルなオブジェクトには、数値、文字列、およびタプルなどがあります。これらのオブジェクトは値を変えられません。別の値を記憶させる際には、新たなオブジェクトを作成しなければなりません。イミュータブルなオブジェクトは、固定のハッシュ値が必要となる状況で重要な役割を果たします。辞書のキーがその例です。

import path

path based finder が import するモジュールを検索する場所 (または *path entry*) のリスト。import 中、このリストは通常 `sys.path` から来ますが、サブパッケージの場合は親パッケージの `__path__` 属性からも来ます。

importing

あるモジュールの Python コードが別のモジュールの Python コードで使えるようにする処理です。

importer

モジュールを探してロードするオブジェクト。*finder* と *loader* のどちらでもあるオブジェクト。

interactive

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see tut-interac.

interpreted

Python はインタプリタ形式の言語であり、コンパイラ言語の対極に位置します。(バイトコードコンパイラがあるために、この区別は曖昧ですが。) ここでのインタプリタ言語とは、ソースコードのファイルを、まず実行可能形式にしてから実行させるといった操作なしに、直接実行できることを意味します。インタプリタ形式の言語は通常、コンパイラ形式の言語よりも開発／デバッグのサイクルは短いものの、プログラムの実行は一般に遅いです。**対話的** も参照してください。

interpreter shutdown

Python インタープリターはシャットダウンを要請された時に、モジュールやすべてのクリティカルな内部構造をなどの、すべての確保したリソースを段階的に開放する、特別なフェーズに入ります。このフェーズは **ガベージコレクタ** を複数回呼び出します。これによりユーザー定義のデストラクターや weakref コールバックが呼び出されることがあります。シャットダウンフェーズ中に実行されるコードは、それが依存するリソースがすでに機能しない (よくある例はライブラリーモジュールや warning 機構です) ために様々な例外に直面します。

インタープリタがシャットダウンする主な理由は `__main__` モジュールや実行されていたスクリプトの実行が終了したことです。

iterable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a *`__getitem__()`* method that implements *sequence* semantics.

Iterables can be used in a *`for`* loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The *`for`* statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator

データの流れを表現するオブジェクトです。イテレータの `__next__()` メソッドを繰り返し呼び出す (または組み込み関数 `next()` に渡す) と、流れの中の要素を一つずつ返します。データがなくなると、代わりに `StopIteration` 例外を送出します。その時点で、イテレータオブジェクトは尽きており、それ以降は `__next__()` を何度呼んでも `StopIteration` を送出します。イテレータは、そのイテレータオブジェクト自体を返す `__iter__()` メソッドを実装しなければならないので、イテレータは他の iterable を受理するほとんどの場所で利用できます。はっきりとした例外は複数の反復を行うようなコードです。(`list` のような) コンテナオブジェクトは、自身を `iter()` 関数にオブジェクトに渡したり *`for`* ループ内で使うたびに、新たな未使用のイテレータを生成します。これをイテレータで行おうとすると、前回のイテレーションで使用済みの同じイテレータオブジェクトを単純に返すため、空のコンテナのようになってしまします。

詳細な情報は typeiter にあります。

CPython 実装の詳細: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

key function

(キー関数) キー関数、あるいは照合関数とは、ソートや順序比較のための値を返す呼び出し可能オブジェクト (callable) です。例えば、`locale.strxfrm()` をキー関数に使えば、ロケール依存のソートの慣習にのっとったソートキーを返します。

Python の多くのツールはキー関数を受け取り要素の並び順やグループ化を管理します。`min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, `itertools.groupby()` 等があります。

キー関数を作る方法はいくつかあります。例えば `str.lower()` メソッドを大文字小文字を区別しないソートを行うキー関数として使うことが出来ます。あるいは、`lambda r: (r[0], r[2])` のような *`lambda`*

式からキー関数を作ることができます。また、`operator.attrgetter()`, `operator.itemgetter()`, `operator.methodcaller()` の 3 つのキー関数コンストラクタがあります。キー関数の作り方と使い方の例は Sorting HOW TO を参照してください。

keyword argument 実
: 引数 を参照してください。

lambda
: (ラムダ) 無名のインライン関数で、関数が呼び出されたときに評価される 1 つの 式 を含みます。ラムダ関数を作る構文は `lambda [parameters]: expression` です。

LBYL
: 「ころばぬ先の杖 (look before you leap)」の略です。このコーディングスタイルでは、呼び出しや検索を行う前に、明示的に前提条件 (pre-condition) 判定を行います。*EAFP* アプローチと対照的で、*if* 文がたくさん使われるのが特徴的です。

  マルチスレッド化された環境では、LBYL アプローチは ” 見る” 過程と ” 飛ぶ” 過程の競合状態を引き起こすリスクがあります。例えば、`if key in mapping: return mapping[key]` というコードは、判定の後、別のスレッドが探索の前に *mapping* から *key* を取り除くと失敗します。この問題は、ロックするか EAFP アプローチを使うことで解決できます。

list A
: built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

list comprehension
: (リスト内包表記) シーケンス中の全てあるいは一部の要素を処理して、その結果からなるリストを返す、コンパクトな方法です。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` とすると、0 から 255 までの偶数を 16 進数表記 (0x..) した文字列からなるリストを生成します。*if* 節はオプションです。*if* 節がない場合、`range(256)` の全ての要素が処理されます。

loader モ
: ジュールをロードするオブジェクト。`load_module()` という名前のメソッドを定義していなければなりません。ローダーは一般的に *finder* から返されます。詳細は **PEP 302** を、*abstract base class* については `importlib.abc.Loader` を参照してください。

ロケールエンコーディング
: On Unix, it is the encoding of the LC_CTYPE locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

  On Windows, it is the ANSI code page (ex: `"cp1252"`).

  On Android and VxWorks, Python uses `"utf-8"` as the locale encoding.

  `locale.getencoding()` can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

magic method
: *special method* のくだけた同義語です。

mapping
: (マッピング) 任意のキー探索をサポートしていて、`collections.abc.Mapping` か `collections.abc.MutableMapping` の 抽象基底クラス で指定されたメソッドを実装しているコンテナオブジェクトです。例えば、`dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` などです。

meta path finder
: `sys.meta_path` を検索して得られた *finder*. meta path finder は *path entry finder* と関係はありますが、別物です。

  meta path finder が実装するメソッドについては `importlib.abc.MetaPathFinder` を参照してください。

metaclass
: (メタクラス) クラスのクラスです。クラス定義は、クラス名、クラスの辞書と、基底クラスのリストを作ります。メタクラスは、それら 3 つを引数として受け取り、クラスを作る責任を負います。ほとんどのオブジェクト指向言語は (訳注: メタクラスの) デフォルトの実装を提供しています。Python が特別なのはカスタムのメタクラスを作成できる点です。ほとんどのユーザーに取って、メタクラスは全く必要のないものです。しかし、一部の場面では、メタクラスは強力でエレガントな方法を提供します。たとえば属性アクセスのログを取ったり、スレッドセーフ性を追加したり、オブジェクトの生成を追跡したり、シングルトンを実装するなど、多くの場面で利用されます。

  詳細は **メタクラス** を参照してください。

メソッド
: (メソッド) クラス本体の中で定義された関数。そのクラスのインスタンスの属性として呼び出された場合、メソッドはインスタンスオブジェクトを第一 **引数** として受け取ります (この第一引数は通常 `self` と呼ばれます)。**関数** と **ネストされたスコープ** も参照してください。

method resolution order
: Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

module
: (モジュール) Python コードの組織単位としてはたらくオブジェクトです。モジュールは任意の Python オブジェクトを含む名前空間を持ちます。モジュールは *importing* の処理によって Python に読み込まれます。

  パッケージ を参照してください。

module spec
: モ

ジュールをロードするのに使われるインポート関連の情報を含む名前空間です。`importlib.machinery.ModuleSpec` のインスタンスです。

MRO

method resolution order を参照してください。

mutable

(ミュータブル) ミュータブルなオブジェクトは、`id()` を変えることなく値を変更できます。**イミュータブル**) も参照してください。

named tuple ”

名前付きタプル” という用語は、タプルを継承していて、インデックスが付く要素に対し属性を使ってのアクセスもできる任意の型やクラスに応用されています。その型やクラスは他の機能も持っていることもあります。

`time.localtime()` や `os.stat()` の返り値を含むいくつかの組み込み型は名前付きタプルです。他の例は `sys.float_info` です:

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

namespace

(名前空間) 変数が格納される場所です。名前空間は辞書として実装されます。名前空間にはオブジェクトの (メソッドの) 入れ子になったものだけでなく、局所的なもの、大域的なもの、そして組み込みのものがあります。名前空間は名前の衝突を防ぐことによってモジュール性をサポートする。例えば関数 `builtins.open` と `os.open()` は名前空間で区別されています。また、どのモジュールが関数を実装しているか明示することによって名前空間は可読性と保守性を支援します。例えば、`random.seed()` や `itertools.islice()` と書くと、それぞれモジュール `random` や `itertools` で実装されていることが明らかです。

namespace package

(名前空間パッケージ) サブパッケージのコンテナとしてのみ提供される **PEP 420** で定義された *package* です。名前空間パッケージは物理的な表現を持たないことができ、`__init__.py` ファイルを持たないため、*regular package* とは異なります。

module を参照してください。

nested scope
: (ネストされたスコープ) 外側で定義されている変数を参照する機能です。例えば、ある関数が別の関数の中で定義されている場合、内側の関数は外側の関数中の変数を参照できます。ネストされたスコープはデフォルトでは変数の参照だけができ、変数の代入はできないので注意してください。ローカル変数は、最も内側のスコープで変数を読み書きします。同様に、グローバル変数を使うとグローバル名前空間の値を読み書きします。*nonlocal* で外側の変数に書き込めます。

new-style class
: Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like *__slots__*, descriptors, properties, *__getattribute__()*, class methods, and static methods.

object
: (オブジェクト) 状態 (属性や値) と定義された振る舞い (メソッド) をもつ全てのデータ。もしくは、全ての **新スタイルクラス** の究極の基底クラスのこと。

optimized scope
: A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package
: (パッケージ) サブモジュールや再帰的にサブパッケージを含むことの出来る *module* のことです。専門的には、パッケージは `__path__` 属性を持つ Python オブジェクトです。

  regular package と *namespace package* を参照してください。

parameter
: (仮引数) 名前付の実体で **関数** (や **メソッド**) の定義において関数が受ける **実引数** を指定します。仮引数には 5 種類あります:

  - **位置またはキーワード**: **位置** であるいは **キーワード引数** として渡すことができる引数を指定します。これはたとえば以下の *foo* や *bar* のように、デフォルトの仮引数の種類です:

    ```
    def func(foo, bar=None): ...
    ```

  - 位置専用: 位置によってのみ与えられる引数を指定します。位置専用の引数は 関数定義の引数のリストの中でそれらの後ろに `/` を含めることで定義できます。例えば下記の *posonly1* と *posonly2* は位置専用引数になります:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- **キーワード専用**: キーワードによってのみ与えられる引数を指定します。キーワード専用の引数を定義できる場所は、例えば以下の *kw_only1* や *kw_only2* のように、関数定義の仮引数リストに含めた可変長位置引数または裸の `*` の後です:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- **可変長位置**: (他の仮引数で既に受けられた任意の位置引数に加えて) 任意の個数の位置引数が与えられることを指定します。このような仮引数は、以下の *args* のように仮引数名の前に `*` をつけることで定義できます:

```
def func(*args, **kwargs): ...
```

- **可変長キーワード**: (他の仮引数で既に受けられた任意のキーワード引数に加えて) 任意の個数のキーワード引数が与えられることを指定します。このような仮引数は、上の例の *kwargs* のように仮引数名の前に `**` をつけることで定義できます。

仮引数はオプションと必須の引数のどちらも指定でき、オプションの引数にはデフォルト値も指定できます。

仮引数 、FAQ の 実引数と仮引数の違いは何ですか? 、`inspect.Parameter` クラス、**関数定義** セクション、**PEP 362** を参照してください。

path entry
: *path based finder* が import するモジュールを探す *import path* 上の 1 つの場所です。

path entry finder
: `sys.path_hooks` にある callable (つまり *path entry hook*) が返した *finder* です。与えられた *path entry* にあるモジュールを見つける方法を知っています。

  パスエントリーファインダが実装するメソッドについては `importlib.abc.PathEntryFinder` を参照してください。

path entry hook
: A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

path based finder
: デフォルトの *meta path finder* の 1 つは、モジュールの *import path* を検索します。

path-like object
: (path-like オブジェクト) ファイルシステムパスを表します。path-like オブジェクトは、パスを表す `str` オブジェクトや `bytes` オブジェクト、または `os.PathLike` プロトコルを実装したオブジェクトのどれかです。`os.PathLike` プロトコルをサポートしているオブジェクトは `os.fspath()` を呼び出すことで `str`

または bytes のファイルシステムパスに変換できます。os.fsdecode() と os.fsencode() はそれぞれ str あるいは bytes になるのを保証するのに使えます。**PEP 519** で導入されました。

PEP
: Python Enhancement Proposal。PEP は、Python コミュニティに対して情報を提供する、あるいは Python の新機能やその過程や環境について記述する設計文書です。PEP は、機能についての簡潔な技術的仕様と提案する機能の論拠 (理論) を伝えるべきです。

  PEP は、新機能の提案にかかる、コミュニティによる問題提起の集積と Python になされる設計決断の文書化のための最上位の機構となることを意図しています。PEP の著者にはコミュニティ内の合意形成を行うこと、反対意見を文書化することの責務があります。

  PEP 1 を参照してください。

portion
: **PEP 420** で定義されている、namespace package に属する、複数のファイルが (zip ファイルに格納されている場合もある) 1 つのディレクトリに格納されたもの。

位置引数 (positional argument) 実
: 引数 を参照してください。

provisional API
: (暫定 API) 標準ライブラリの後方互換性保証から計画的に除外されたものです。そのようなインターフェースへの大きな変更は、暫定であるとされている間は期待されていませんが、コア開発者によって必要とみなされれば、後方非互換な変更 (インターフェースの削除まで含まれる) が行われえます。このような変更はむやみに行われるものではありません -- これは API を組み込む前には見落とされていた重大な欠陥が露呈したときにのみ行われます。

  暫定 API についても、後方互換性のない変更は「最終手段」とみなされています。問題点が判明した場合でも後方互換な解決策を探すべきです。

  このプロセスにより、標準ライブラリは問題となるデザインエラーに長い間閉じ込められることなく、時代を超えて進化を続けられます。詳細は **PEP 411** を参照してください。

provisional package
: *provisional API* を参照してください。

Python 3000
: Python 3.x リリースラインのニックネームです。(Python 3 が遠い将来の話だった頃に作られた言葉です。) "Py3k" と略されることもあります。

Pythonic 他
: の言語で一般的な考え方で書かれたコードではなく、Python の特に一般的なイディオムに従った考え方やコード片。例えば、Python の一般的なイディオムでは *for* 文を使ってイテラブルのすべての要素に渡ってループします。他の多くの言語にはこの仕組みはないので、Python に慣れていない人は代わりに数値のカウンターを使うかもしれません:

```
for i in range(len(food)):
    print(food[i])
```

これに対し、きれいな Pythonic な方法は:

```
for piece in food:
    print(piece)
```

qualified name

(修飾名) モジュールのグローバルスコープから、そのモジュールで定義されたクラス、関数、メソッドへの、”パス”を表すドット名表記です。**PEP 3155** で定義されています。トップレベルの関数やクラスでは、修飾名はオブジェクトの名前と同じです:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

モジュールへの参照で使われると、**完全修飾名** *(fully qualified name)* はすべての親パッケージを含む全体のドット名表記、例えば `email.mime.text` を意味します:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count

(参照カウント) あるオブジェクトに対する参照の数。参照カウントが 0 になったとき、そのオブジェクトは破棄されます。**永続** であり、参照カウントが決して変更されないために割り当てが解除されないオブジェクトもあります。参照カウントは通常は Python のコード上には現れませんが、*CPython* 実装の重要な要素です。プログラマーは、任意のオブジェクトの参照カウントを知るために `sys.getrefcount()` 関数を呼び出すことが出来ます。

regular package 伝

統的な、`__init__.py` ファイルを含むディレクトリとしての *package*。

namespace package を参照してください。

REPL

”read – eval – print loop” の頭字語で、**対話型** インタープリターシェルの別名。

__slots__
: クラス内での宣言で、インスタンス属性の領域をあらかじめ定義しておき、インスタンス辞書を排除することで、メモリを節約します。これはよく使われるテクニックですが、正しく扱うには少しトリッキーなので、稀なケース、例えばメモリが死活問題となるアプリケーションでインスタンスが大量に存在する、といったときを除き、使わないのがベストです。

sequence
: An *iterable* which supports efficient element access using integer indices via the *__getitem__()* special method and defines a *__len__()* method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports *__getitem__()* and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

  The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just *__getitem__()* and *__len__()*, adding `count()`, `index()`, *__contains__()*, and *__reversed__()*. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

set comprehension
: (集合内包表記) iterable 内の全てあるいは一部の要素を処理して、その結果からなる集合を返すコンパクトな書き方です。`results = {c for c in 'abracadabra' if c not in 'abc'}` とすると、`{'r', 'd'}` という文字列の辞書を生成します。**リスト、集合、辞書の表示** を参照してください。

single dispatch
: *generic function* の一種で実装は一つの引数の型により選択されます。

slice
: (スライス) 一般に **シーケンス** の一部を含むオブジェクト。スライスは、添字表記 `[]` で与えられた複数の数の間にコロンを書くことで作られます。例えば、`variable_name[1:3:5]` です。角括弧 (添字) 記号は `slice` オブジェクトを内部で利用しています。

soft deprecated
: A soft deprecation can be used when using an API which should no longer be used to write new code, but it remains safe to continue using it in existing code. The API remains documented and tested, but will not be developed further (no enhancement).

  The main difference between a "soft" and a (regular) "hard" deprecation is that the soft deprecation does not imply scheduling the removal of the deprecated API.

  Another difference is that a soft deprecation does not issue a warning.

  See PEP 387: Soft Deprecation.

special method

(特殊メソッド) ある型に特定の操作、例えば加算をするために Python から暗黙に呼び出されるメソッド。この種類のメソッドは、メソッド名の最初と最後にアンダースコア 2 つがついています。特殊メソッドについては **特殊メソッド名** で解説されています。

statement

(文) 文はスイート (コードの" ブロック") に不可欠な要素です。文は **式** かキーワードから構成されるもののどちらかです。後者には *if*、*while*、*for* があります。

static type checker

An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

strong reference

In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

See also *borrowed reference*.

text encoding

A string in Python is a sequence of Unicode code points (in range `U+0000--U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as "encoding", and recreating the string from the sequence of bytes is known as "decoding".

There are a variety of different text serialization codecs, which are collectively referred to as "text encodings".

text file

(テキストファイル) `str` オブジェクトを読み書きできる *file object* です。しばしば、テキストファイルは実際にバイト指向のデータストリームにアクセスし、**テキストエンコーディング** を自動的に行います。テキストファイルの例は、`sys.stdin`, `sys.stdout`, `io.StringIO` インスタンスなどをテキストモード (`'r'` or `'w'`) で開いたファイルです。

bytes-like **オブジェクト** を読み書きできるファイルオブジェクトについては、**バイナリファイル** も参照してください。

triple-quoted string

(三重クォート文字列) 3 つの連続したクォート記号 (") かアポストロフィー (') で囲まれた文字列。通常の (一重) クォート文字列に比べて表現できる文字列に違いはありませんが、幾つかの理由で有用です。1 つか

2 つの連続したクォート記号をエスケープ無しに書くことができますし、行継続文字 (\) を使わなくても複数行にまたがることができるので、ドキュメンテーション文字列を書く時に特に便利です。

type
: (型) Python オブジェクトの型はオブジェクトがどのようなものかを決めます。あらゆるオブジェクトは型を持っています。オブジェクトの型は `__class__` 属性でアクセスしたり、`type(obj)` で取得したり出来ます。

type alias
: (型エイリアス) 型の別名で、型を識別子に代入して作成します。

  型エイリアスは **型ヒント** を単純化するのに有用です。例えば:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

  これは次のようにより読みやすくできます:

```
Color = tuple[int, int, int]

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

  機能の説明がある `typing` と **PEP 484** を参照してください。

type hint
: (型ヒント) 変数、クラス属性、関数のパラメータや返り値の期待される型を指定する *annotation* です。

  Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

  グローバル変数、クラス属性、関数で、ローカル変数でないものの型ヒントは `typing.get_type_hints()` で取得できます。

  機能の説明がある `typing` と **PEP 484** を参照してください。

universal newlines
: テキストストリームの解釈法の一つで、以下のすべてを行末と認識します: Unix の行末規定 `'\n'`、Windows の規定 `'\r\n'`、古い Macintosh の規定 `'\r'`。利用法について詳しくは、**PEP 278** と **PEP 3116** 、さらに `bytes.splitlines()` も参照してください。

variable annotation
: (変数アノテーション) 変数あるいはクラス属性の *annotation* 。

  変数あるいはクラス属性に注釈を付けたときは、代入部分は任意です:

```
class C:
    field: 'annotation'
```

変数アノテーションは通常は **型ヒント** のために使われます: 例えば、この変数は `int` の値を取ることを期待されています:

```
count: int = 0
```

変数アノテーションの構文については **注釈付き代入文** *(annotated assignment statements)* 節で解説しています。

機能の説明がある *function annotation*, **PEP 484** , **PEP 526** を参照してください。また、アノテーションを利用するベストプラクティスとして annotations-howto も参照してください。

virtual environment
: (仮想環境) 協調的に切り離された実行環境です。これにより Python ユーザとアプリケーションは同じシステム上で動いている他の Python アプリケーションの挙動に干渉することなく Python パッケージのインストールと更新を行うことができます。

  `venv` を参照してください。

virtual machine
: (仮想マシン) 完全にソフトウェアにより定義されたコンピュータ。Python の仮想マシンは、バイトコードコンパイラが出力した **バイトコード** を実行します。

Zen of Python
: (Python の悟り) Python を理解し利用する上での導きとなる、Python の設計原則と哲学をリストにしたものです。対話プロンプトで "`import this`" とするとこのリストを読めます。

付録

B

このドキュメントについて

このドキュメントは、Python のドキュメントを主要な目的として作られた ドキュメントプロセッサの Sphinx を利用して、reStructuredText 形式のソースから生成されました。

ドキュメントとそのツール群の開発は、Python 自身と同様に完全にボランティアの努力です。もしあなたが貢献したいなら、どのようにすればよいかについて reporting-bugs ページをご覧下さい。新しいボランティアはいつでも歓迎です!（訳注: 日本語訳の問題については、GitHub 上の Issue Tracker で報告をお願いします。）

多大な感謝を:

- Fred L. Drake, Jr., オリジナルの Python ドキュメントツールセットの作成者で、ドキュメントの多くを書きました。
- Docutils プロジェクト reStructuredText と Docutils ツールセットを作成しました。
- Fredrik Lundh の Alternative Python Reference プロジェクトから Sphinx は多くのアイデアを得ました。

B.1 Python ドキュメント 貢献者

多くの方々が Python 言語、Python 標準ライブラリ、そして Python ドキュメンテーションに貢献してくれています。ソース配布物の Misc/ACKS に、それら貢献してくれた人々を部分的にではありますがリストアップしてあります。

Python コミュニティからの情報提供と貢献がなければこの素晴らしいドキュメンテーションは生まれませんでした -- ありがとう!

付録

C

歴史とライセンス

C.1 Python の歴史

Python は 1990 年代の始め、オランダにある Stichting Mathematisch Centrum (CWI, https://www.cwi.nl/ 参照) で Guido van Rossum によって ABC と呼ばれる言語の後継言語として生み出されました。その後多くの人々が Python に貢献していますが、Guido は今日でも Python 製作者の先頭に立っています。

1995 年、Guido は米国ヴァージニア州レストンにある Corporation for National Reserch Initiatives (CNRI, https://www.cnri.reston.va.us/ 参照) で Python の開発に携わり、いくつかのバージョンをリリースしました。

2000 年 3 月、Guido と Python のコア開発チームは BeOpen.com に移り、BeOpen PythonLabs チームを結成しました。同年 10 月、PythonLabs チームは Digital Creations (現在の Zope Corporation, https://www.zope.org/ 参照) に移りました。そして 2001 年、Python に関する知的財産を保有するための非営利組織 Python Software Foundation (PSF、https://www.python.org/psf/ 参照) を立ち上げました。このとき Zope Corporation は PSF の賛助会員になりました。

Python のリリースは全てオープンソース (オープンソースの定義は https://opensource.org/ を参照してください) です。歴史的にみて、ごく一部を除くほとんどの Python リリースは GPL 互換になっています; 各リリースについては下表にまとめてあります。

リリース	ベース	西暦年	権利	GPL 互換
0.9.0 - 1.2	n/a	1991-1995	CWI	yes
1.3 - 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 以降	2.1.1	2001-現在	PSF	yes

注釈

「GPL 互換」という表現は、Python が GPL で配布されているという意味ではありません。Python のライセンスは全て、GPL と違い、変更したバージョンを配布する際に変更をオープンソースにしなくてもかまいません。GPL 互換のライセンスの下では、GPL でリリースされている他のソフトウェアと Python を組み合わせられますが、それ以外のライセンスではそうではありません。

Guido の指示の下、これらのリリースを可能にしてくださった多くのボランティアのみなさんに感謝します。

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

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.13.0rc2

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using Python
   3.13.0rc2 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 3.13.0rc2 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python 3.13.0rc2 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 3.13.0rc2 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
   3.13.0rc2.

4. PSF is making Python 3.13.0rc2 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 3.13.0rc2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.13.0rc2
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.13.0rc2, 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 3.13.0rc2, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

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

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

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

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

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

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

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

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

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

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
```

(次のページに続く)

(前のページからの続き)

```
   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 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0rc2 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
```

(次のページに続く)

(前のページからの続き)

```
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 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` C extension underlying the `random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

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

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

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

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

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

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

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

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
```

(次のページに続く)

(前のページからの続き)

```
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 ソケット

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

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

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

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

C.3.3 Asynchronous socket services

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

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

C.3.4 Cookie management

The `http.cookies` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

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

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
```

(次のページに続く)

(前のページからの続き)

```
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 Execution tracing

The `trace` module contains the following notice:

```
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 UUencode and UUdecode functions

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
```

(次のページに続く)

(前のページからの続き)

```
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

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

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
```

(次のページに続く)

(前のページからの続き)

```
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test.test_epoll` module contains the following notice:

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

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

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

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

C.3.9 Select kqueue

`select` モジュールは kqueue インターフェースについての次の告知を含んでいます:

```
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
```

(次のページに続く)

(前のページからの続き)

```
   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

The file `Python/pyhash.c` contains Marek Majkowski' implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<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 と dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

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

C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
```

(次のページに続く)

(前のページからの続き)

```
the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all
other entities that control, are controlled by, or are under common
control with that entity. For the purposes of this definition,
"control" means (i) the power, direct or indirect, to cause the
direction or management of such entity, whether by contract or
otherwise, or (ii) ownership of fifty percent (50%) or more of the
outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity
exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications,
including but not limited to software source code, documentation
source, and configuration files.

"Object" form shall mean any form resulting from mechanical
transformation or translation of a Source form, including but
not limited to compiled object code, generated documentation,
and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or
Object form, made available under the License, as indicated by a
copyright notice that is included in or attached to the work
(an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object
form, that is based on (or derived from) the Work and for which the
editorial revisions, annotations, elaborations, or other modifications
represent, as a whole, an original work of authorship. For the purposes
of this License, Derivative Works shall not include works that remain
separable from, or merely link (or bind by name) to the interfaces of,
the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including
the original version of the Work and any modifications or additions
to that Work or Derivative Works thereof, that is intentionally
submitted to Licensor for inclusion in the Work by the copyright owner
or by an individual or Legal Entity authorized to submit on behalf of
the copyright owner. For the purposes of this definition, "submitted"
means any form of electronic, verbal, or written communication sent
to the Licensor or its representatives, including but not limited to
communication on electronic mailing lists, source code control systems,
and issue tracking systems that are managed by, or on behalf of, the
Licensor for the purpose of discussing and improving the Work, but
excluding communication that is conspicuously marked or otherwise
```

(次のページに続く)

(前のページからの続き)

```
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
```

(次のページに続く)

(前のページからの続き)

```
      distribution, then any Derivative Works that You distribute must
      include a readable copy of the attribution notices contained
      within such NOTICE file, excluding those notices that do not
      pertain to any part of the Derivative Works, in at least one
      of the following places: within a NOTICE text file distributed
      as part of the Derivative Works; within the Source form or
      documentation, if provided along with the Derivative Works; or,
      within a display generated by the Derivative Works, if and
      wherever such third-party notices normally appear. The contents
      of the NOTICE file are for informational purposes only and
      do not modify the License. You may add Your own attribution
      notices within Derivative Works that You distribute, alongside
      or as an addendum to the NOTICE text from the Work, provided
      that such additional attribution notices cannot be construed
      as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
```

(次のページに続く)

(前のページからの続き)

```
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

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

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

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

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(次のページに続く)

(前のページからの続き)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

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

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

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

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

C.3.15 zlib

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

```
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.
```

(次のページに続く)

(前のページからの続き)

```
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

The implementation of the hash table used by the `tracemalloc` で使用しているハッシュテーブルの実装は、cfuhash プロジェクトのものに基づきます:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

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

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

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

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
```

(次のページに続く)

(前のページからの続き)

```
"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

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

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

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

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

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

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

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

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

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

C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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:
```

(次のページに続く)

(前のページからの続き)

```
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.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

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

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

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

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

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 unmodified, 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 ``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 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.
```

付録

D

COPYRIGHT

ライセンスおよび許諾に関する完全な情報は、**歴史とライセンス** を参照してください。

索引

アルファベット以外

A

B

C

D

E

F

W

X

Y

Z