

---

# Python リファレンスマニュアル

リリース 2.3.4

Guido van Rossum

Fred L. Drake, Jr., editor

日本語訳: Python ドキュメント翻訳プロジェクト

平成 17 年 3 月 29 日

**PythonLabs**

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

Copyright © 2001, 2002, 2003 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

Translation Copyright © 2003 Python Document Japanese Translation Project. All rights reserved.

ライセンスおよび許諾に関する完全な情報は、このドキュメントの末尾を参照してください。

## 概要

Python はインタプリタ形式の、オブジェクト指向な高レベルプログラミング言語で、動的なセマンティクスを持っています。Python の高レベルな組み込みデータ構造は、動的な型付け機能や動的な結合機能と組み合わせることで、迅速なアプリケーション開発や既存のソフトウェアコンポーネント間をつなぐスクリプト言語、または糊 (glue) 言語として Python を魅力的な存在にしています。Python は単純で学びやすい文法なので、可読性が高まり、プログラムのメンテナンスにかかるコストを低減します。Python は、プログラムのモジュール化や再利用を助けるモジュールとパッケージをサポートします。Python インタプリタと多数の標準ライブラリは、ほとんどのプラットフォームでソースコード形式でもバイナリ形式でも無料で入手することができ、無料で配布することができます。

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、的確かつ完璧な記述を目指しています。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、*Python* ライブラリリファレンス で述べられています。形式ばらない Python 言語入門には、*Python* チュートリアル 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: *Python* インタプリタの拡張と埋め込みでは、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、*Python/C API* リファレンスマニュアルでは、C/C++ プログラマが利用できるインタフェースについて詳細に記述しています。



# 目次

第 1 章	導入	1
1.1	本マニュアルにおける表記法	1
第 2 章	字句解析	3
2.1	行構造	3
2.2	その他のトークン	6
2.3	識別子 (identifier) およびキーワード (keyword)	6
2.4	リテラル (literal)	7
2.5	演算子 (operator)	11
2.6	デリミタ (delimiter)	12
第 3 章	データモデル	13
3.1	オブジェクト、値、および型	13
3.2	標準型の階層	14
3.3	特殊メソッド名	23
第 4 章	実行モデル	37
4.1	名前づけと束縛 (naming and binding)	37
4.2	例外	39
第 5 章	式 (expression)	41
5.1	算術変換 (arithmetic conversion)	41
5.2	アトム、原子的要素 (atom)	41
5.3	一次語 (primary)	44
5.4	べき乗演算 (power operator)	47
5.5	単項算術演算 (unary arithmetic operation)	48
5.6	二項算術演算 (binary arithmetic operation)	48
5.7	シフト演算 (shifting operation)	49
5.8	ビット単位演算の二項演算 (binary bit-wise operation)	49
5.9	比較 (comparison)	49
5.10	ブール演算 (boolean operation)	51
5.11	ラムダ (lambda)	52
5.12	式のリスト	52
5.13	評価順序	52
5.14	まとめ	52

<b>第 6 章</b>	<b>単純文 (simple statement)</b>	<b>55</b>
6.1	式文 (expression statement)	55
6.2	Assert 文 (assert statement)	55
6.3	代入文 (assignment statement)	56
6.4	pass 文	58
6.5	del 文	59
6.6	print 文	59
6.7	return 文	60
6.8	yield 文	60
6.9	raise 文	61
6.10	break 文	61
6.11	continue 文	61
6.12	import 文	62
6.13	global 文	64
6.14	exec 文	64
<b>第 7 章</b>	<b>複合文 (compound statement)</b>	<b>67</b>
7.1	if 文	68
7.2	while 文	68
7.3	for 文	68
7.4	try 文	69
7.5	関数定義	70
7.6	クラス定義	71
<b>第 8 章</b>	<b>トップレベル要素</b>	<b>73</b>
8.1	完全な Python プログラム	73
8.2	ファイル入力	73
8.3	対話的入力	74
8.4	式入力	74
<b>付 録 A</b>	<b>歴史とライセンス</b>	<b>75</b>
A.1	History of the software	75
A.2	Terms and conditions for accessing or otherwise using Python	76
<b>付 録 B</b>	<b>日本語訳について</b>	<b>79</b>
B.1	このドキュメントについて	79
B.2	翻訳者一覧 (敬称略)	79
<b>索引</b>		<b>81</b>

# 導入

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れ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ない、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、広く使われている Python 実装は現在のところ唯一 (今や第二の実装が存在しますが!) 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 (implementation 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。これらはここではドキュメント化されていませんが、*Python* ライブラリリファレンス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わりをもっているときについて触れています。

## 1.1 本マニュアルにおける表記法

字句解析と構文に関する記述では、BNF 文法記法に手を加えたものを使っています。この記法では、以下のような記述形式をとります:

```
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) とコロニー一つから始まります。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。



# 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、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 7-bit の ASCII 文字セットをプログラムのテキストに使用します。2.3 で追加された仕様: エンコード宣言を使って、文字列リテラルやコメントに ASCII ではない文字セットが使われていることを明示できます。以前のバージョンとの互換性のために、Python は 8-bit 文字が見つかったとしても警告を出すだけにとどめます; こうした警告は、エンコーディングを明示したり、バイナリデータの場合には文字ではなくエスケープシーケンスを使うことで解決できます。

実行時の文字セットは、プログラムが接続されている I/O デバイスにもよりますが、通常 ASCII のサブセットです。

将来のバージョンとの互換性に関する注意: 8-bit 文字に対する文字セットを ISO Latin-1 (ラテン語系アルファベットを用いるほとんどの西欧言語をカバーする ASCII の上位セット) とみなしたい気にもなるかもしれませんが、おそらく Unicode を編集できるテキストエディタが将来一般的になるはずですが、こうしたエディタでは一般的に UTF-8 エンコードを使用しますが、UTF-8 エンコードは ASCII の上位セットではあるものの、文字序数 (ordinal) 128-255 の扱いが非常に異なります。この問題に関してはまだ合意が得られていませんが、Latin-1 と UTF-8 のどちらかとみなすのは、たとえ現在の実装が Latin-1 びいきのように思えたとしても賢明とはいえません。これはソースコード文字セットと実行時の文字セットのどちらにも該当します。

## 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、利用しているプラットフォームで行終端として取り決めている文字列で終端されたものです。UNIX では、行終端は ASCII LF (行送り: *linefeed*) 文字です。Windows では、ASCII 配列の CR LF (復帰: *return* に続いて行送り) です。Macintosh では、ASCII CR (復帰) 文字です。

### 2.1.3 コメント

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; コメントはトークンになりません。

### 2.1.4 エンコード宣言 (encoding declaration)

Python スクリプト中の最初の行か、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 (encoding declaration) として処理されます; 表現に対する最初のマッチ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。エンコード宣言式として推奨する形式は、GNU Emacs が認識できる形式

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

または、Bram Moolenaar による VIM が認識できる形式

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

です。さらに、ファイルの先頭のバイト列が UTF-8 バイトオーダー記号 (`'\xef\xbb\xbf'`) の場合、ファイルのエンコードは UTF-8 と宣言されているものとします (この機能は Microsoft の **notepad** やその他のエディタでサポートされています)。

エンコードが宣言されている場合、Python はそのエンコード名を認識できなければなりません。宣言されたエンコードは全ての字句解析、特に文字列の終端を検出する際や Unicode リテラルの内容を翻訳する上で用いられます。文字列リテラルは文法的な解析を行うために Unicode に変換され、解釈が行われる前に元のエンコードに戻されます。エンコード宣言は宣言全体が一行に収まっていなければなりません。

### 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
```

となります。

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かもしれません。標準的な実装では、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、複数行からなる実行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。

まず、タブは (左から右の方向に) 1 つから 8 つのスペースで置き換えられ、置き換え後の文字列の終わりの位置までの文字数が 8 の倍数になるように調整されます (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)。次に、空白文字でない最初の文字までのスペースの総数から、その行のインデントを決定します。バックスラッシュを使ってインデントを複数の物理行に分割することはできません; 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。

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

口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:

```

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

```

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。

### 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:

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

識別子 `as` は `import` 文における構文の一部として使われることがありますが、現在のところ予約語ではありません。

将来の Python のバージョンでは、識別子 `as` と `None` はともにキーワードになる予定です。

### 2.3.2 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:

\_\_\* この識別子は `'from module import *'` で `import` されません。対話インタプリタでは、最も最近行われた値評価の結果を記憶するために特殊な識別子 `'_'` が使われます; この識別子は `__builtin__` モジュール内に記憶されます。対話モードでない場合、`'_'` には特殊な意味はなく、定義されていません。6.12 節、“`import` 文”を参照してください。

注意: 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` module を参照してください。

\_\_\_\* システムで定義された (system-defined) 名前です。これらの名前はインタプリタと (標準ライブラリを含む) 実装上で定義されています; アプリケーション側では、この名前規約を使って別の名前を定義しようとすべきではありません。この種の名前のうち、Python で定義されている名前のセットは、将来のバージョンで拡張される可能性があります。3.3 節、“特殊なメソッド名”を参照してください。

\_\_\_\* 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、基底クラスと導出クラスの“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。5.2.1 節、“識別子 (名前)”を参照してください。

## 2.4 リテラル (literal)

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" | "ur" | "R" | "U" | "UR" | "Ur" | "uR"
shortstring   ::= "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring    ::= """ longstringitem* """
               | '"""' longstringitem* '"""'
shortstringitem ::= shortstringchar | escapeseq
longstringitem  ::= longstringchar | escapeseq
shortstringchar ::= <any ASCII character except "\" or newline or the quote>
longstringchar  ::= <any ASCII character except "\">
escapeseq       ::= "\" <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。それは文字列リテラルの `stringprefix` と残りの部分の間に空白を入れてはならないということです。

より平易な説明: 文字列リテラルは、対応する一重引用符 (') または二重引用符 (") で囲われます。また、対応する三連の一重引用符や二重引用符で囲うこともできます (通常、三重クオート文字列: *triple-quoted string* として参照されます)。バックスラッシュ (\) 文字を使って、ある文字を例えば改行文字やバックスラッシュ自体、クオート文字といった別の意味を持つようにエスケープすることができます。文字列リテラルの前には、オプションとして 'r' または 'R' 一文字を接頭してもかまいません; このような文字列は *raw* 文字列 (*raw string*) と呼ばれ、バックスラッシュによるエスケープシーケンスの解釈規則が異なります。'u' や 'U' を接頭すると、文字列は Unicode 文字列 (Unicode string) になります。Unicode 文字列は Unicode コンソーシアムおよび ISO 10646 で定義されている Unicode 文字セットを使います。Unicode 文字列では、文字セットに加えて、以下で説明するようなエスケープシーケンスを利用できます。二つの接頭文字を組み合わせることもできます; この場合、'u' は 'r' より前に出現しなくてはなりません。

三重クオート文字列中には、三連のエスケープされないクオート文字で文字列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)。

'r' または 'R' 接頭文字が見つからないかぎり、文字列中のエスケープシーケンスは標準 C で使われている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	備考
<code>\newline</code>	無視	
<code>\\</code>	バックスラッシュ (\)	
<code>\'</code>	一重引用符 (')	
<code>\"</code>	二重引用符 (")	
<code>\a</code>	ASCII 端末ベル (BEL)	
<code>\b</code>	ASCII バックスペース (BS)	
<code>\f</code>	ASCII フォームフィード (FF)	
<code>\n</code>	ASCII 行送り (LF)	
<code>\N{name}</code>	Unicode データベース中で名前 <i>name</i> を持つ文字 (Unicode のみ)	
<code>\r</code>	ASCII 復帰 (CR)	
<code>\t</code>	ASCII 水平タブ (TAB)	
<code>\uxxxx</code>	16-bit の 16 進数値 <i>xxxx</i> を持つ文字 (Unicode のみ)	(1)
<code>\Uxxxxxxxx</code>	32-bit の 16 進数値 <i>xxxxxxxx</i> を持つ文字 (Unicode のみ)	(2)
<code>\v</code>	ASCII 水平タブ (VT)	
<code>\ooo</code>	8 進数値 <i>ooo</i> を持つ ASCII 文字	(3)
<code>\xhh</code>	16 進数値 <i>hh</i> を持つ ASCII 文字	(4)

備考:

- (1) サロゲートペアの断片を形成する個々のコード単位は、このエスケープシーケンスでエンコードすることができます。
- (2) Unicode 文字はすべてこの方法でエンコードできますが、Python が 16-bit コード単位を扱うようにコンパイルされている (デフォルトの設定です) 場合、基本多言語面 (Basic Multilingual Plane, BMP) 外の文字はサロゲートペア (surrogate pair) を使ってエンコードすることになります。サロゲートペアの断片を形成する個々のコード単位はこのエスケープシーケンスを使ってエンコードすることができます。
- (3) 標準 C と同じく、最大で 3 桁の 8 進数まで受理します。
- (4) 標準 C とは違い、最大で二桁の 16 進数しか受理されません。

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そのまま文字列中に残されます。すなわち、バックスラッシュも文字列中に残ります。(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を誤入力した場合、その結果として出力に失敗しているのが用意にわかります) テーブル中で“(Unicode のみ)”と書かれたエスケープシーケンスは、非 Unicode 文字列リテラル中では認識されないエスケープシーケンスのカテゴリに分類されるので注意してください。

接頭文字 ‘r’ または ‘R’ がある場合、バックスラッシュの後にくる文字はそのまま文字列中に入り、バックスラッシュは全て文字列中に残されます。例えば、文字列リテラル `r"\n"` は二つの文字: バックスラッシュと小文字の ‘n’ からなる文字列を表すことになります。引用符はバックスラッシュでエスケープすることができますが、バックスラッシュ自体も残ってしまいます; 例えば、`r"\\"` は不正でない文字列リテラルで、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"\n"` は正しくない文字列リテラルです (raw 文字列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)。厳密に言えば、(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ため) 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ないということになります。また、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、それら二つの文字として解釈されるので注意してください。

‘r’ および ‘R’ 接頭文字を ‘u’ や ‘U’ と合わせて使った場合、`\uXXXX` エスケープシーケンスは処理されますが、その他のバックスラッシュはすべて文字列中に残されます。例えば、文字列リテラル `ur"\u0062\n"`

は、3 つの Unicode 文字: ‘LATIN SMALL LETTER B’ (ラテン小文字 B)、‘REVERSE SOLIDUS’ (逆向き斜線)、および ‘LATIN SMALL LETTER N’ (ラテン小文字 N) を表します。バックスラッシュの前にバックスラッシュをつけてエスケープすることはできます; しかし、バックスラッシュは両方とも文字列中に残されます。その結果、`\uXXXX` エスケープシーケンスは、バックスラッシュが奇数個連なっている場合にのみ認識されます。

## 2.4.2 文字列リテラルの結合 (concatenation)

複数の文字列リテラルは、互いに異なる引用符を使っても (空白文字で区切って) 隣接させることができ、その意味は各々の文字列を結合したものと同じになります。したがって、`"hello" 'world'` は `"helloworld"` と同じになります。この機能を使うと、長い文字列を分離して、複数行にまたがらせる際に便利です。また、部分文字列ごとにコメントを追加することもできます。例えば:

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

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える (raw 文字列と三重引用符を混ざることさえできます) ので注意してください。

## 2.4.3 数値リテラル

数値リテラルは 4 種類あります: 整数 (plain integer)、長整数 (long integer)、浮動小数点数 (floating point number)、そして虚数 (imaginary number) です。複素数のためのリテラルはありません (複素数は実数と虚数の和で作ることができます)。

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。

## 2.4.4 整数および長整数リテラル

整数および長整数リテラルは以下の字句定義で記述されます:

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

長整数を表す末尾の文字は小文字の `‘l’` でも大文字の `‘L’` でもかまいませんが、`‘l’` は `‘1’` に良く似ているので、常に `‘L’` を使うよう強く勧めます。

整数で表現できる最大の値よりも大きい整数の 10 進表現リテラル (例えば 32-bit 整数を使っている場合には 2147483647) は、長整数として表現できる値であれば受理されます。8 進および 16 進数のリテラルも同様にふるまいますが、整数で表現できる最大の値よりも大きく、かつ (32-bit 算術演算を使う計算機では) 符号無し 32-bit の整数で表現できる最大の値、4294967296 までの間では、符号無し 32-bit 整数と



しての値から 4294967296 を引いて得られる負の整数値になります。値がメモリ上に収まるかどうかという問題を除けば、長整数リテラルには値域の制限がありません。例えば、0xdeadbeef は 32-bit の計算機では -559038737 という値とみなされ、0xdeadbeeffeed は 244837814107885L であるとみなされます。

整数リテラル (最初の行) と長整数リテラル (二行目および三行目) の例を以下に示します:

7	2147483647	0177	0x80000000
3L	79228162514264337593543950336L	0377L	0x100000000L
	79228162514264337593543950336		0xdeadbeeffeed

## 2.4.5 浮動小数点数リテラル

浮動小数点数リテラルは以下の字句定義で記述されます:

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

浮動小数点数における整数部と指数部は 8 進数のように見えることもありますが、10 を基数として解釈されるので注意してください。例えば、'077e010' は正しい表記であり、'77e10' と同じ数を表します。浮動小数点数リテラルの取りうる値の範囲は実装に依存します。浮動小数点数リテラルの例をいくつか示します:

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

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。

## 2.4.6 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 | intpart) ("j" | "J")
```

虚数リテラルは、実数部が 0.0 の複素数を表します。複素数は二つ組の浮動小数点型の数値で表され、それぞれの数値は浮動小数点型と同じ定義域の範囲を持ちます。実数部がゼロでない浮動小数点を生成するには、(3+4j) のように虚数リテラルに浮動小数点数を加算します。以下に虚数リテラルの例をいくつか示します:

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

## 2.5 演算子 (operator)

以下のトークンは演算子です:

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

比較演算子 `<>` と `!=` は、同じ演算子について別の書き方をしたものです。書き方としては `!=` を推奨します; `<>` は時代遅れの書き方です。

## 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:

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

浮動小数点数や虚数リテラル中にピリオドがあってもかまいません。ピリオド三つの列はスライス表記における省略符号 (ellipsis) として特別な意味を持っています。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:

@	\$	?
---	----	---

# 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。)

オブジェクトはアイデンティティ値 (identity)、型 (type)、そして値 (value) を持ちます。オブジェクトが一度生成されると、そのオブジェクトのアイデンティティ値は決して変化することがありません; アイデンティティ値をオブジェクトのメモリ上のアドレスと考えてもかまいません。演算子 `'is'` は、二つのオブジェクト間のアイデンティティ値を比較します; 関数 `id()` は、オブジェクトのアイデンティティ値を表す整数 (現在の実装ではオブジェクトのメモリ上のアドレス) を返します。オブジェクトの型もまた変わることがありません。<sup>1</sup> オブジェクトの型は、そのオブジェクトのサポートする操作 (“長さを持っているか?” など) を決定し、その型のオブジェクトが取りうる値について定義しています。 `type()` 関数は、オブジェクトの型 (型自体も一つのオブジェクトです) を返します。オブジェクトによっては、値 (value) を変えることができます。値を変えられるオブジェクトは変更可能 (*mutable*) であるといひます; 値を一度設定すると、その後は変えることができないオブジェクトは変更不能 (*immutable*) であると呼びます。(変更不能なコンテナオブジェクトが変更可能なオブジェクトへの参照を含んでいる場合、その値は後者のオブジェクトの変更によって変わる場合があります; その場合でも、コンテナの含んでいるオブジェクトの集まりは変わらないため、コンテナは変更不能と考えます。したがって、変更不能性 (immutability) は、厳密には変更できない値を持っていることとは違い、もっと微妙な概念です。) オブジェクトの変更可能性は型で決定されます; 例えば、数値、文字列、およびタプルは変更不能であり、辞書やリストは変更可能です。

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) によって処理されます。実装では、ごみ収集を遅らせたり、全く行わないようにすることができます — 到達可能なオブジェクトをごみ収集処理してしまわないかぎり、どう実装するかは実装品質の問題です。(実装上の注意: 現在の実装では参照カウント (reference-counting) 手順を使っており、(オプションとして) 循環参照を行っているごみ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、循環参照を含むごみ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。循環参照を持つごみオブジェクト収集の制御については、*Python* ライブラリリファレンス を参照してください。)

実装のトレース機能やデバッグ機能を使えば、通常は収集されてしまうようなオブジェクトを生かしておくことがあるので注意してください。また、`'try...except'` 文を使って例外を捕捉できるようにすると、オブジェクトを生かしておくことがあります。

オブジェクトによっては、開かれたファイルやウィンドウといった、“外部 (external) の” リソースに対す

---

<sup>1</sup>Python 2.2 以降、型とクラスの段階的な統合が始まっているため、このドキュメントで主張されている内容が 100% 正確で完全というわけではなくなりました: 例えば、場合によっては、ある管理された条件下でなら、オブジェクトの型を変更することができます。このマニュアルに大幅な改訂が施されるまでは、このドキュメントでの記述は、“旧クラス型 (classic class)” に関してのみ信頼できる内容と考えねばなりません。Python 2.2 および 2.3 では、互換性のためにクラシックなクラスがまだデフォルトとなっています。

る参照を行っています。これらのリソースは、オブジェクトがごみ収集された際に解放されるものと理解されていますが、ごみ収集が行われる保証はないので、こうしたオブジェクトでは外部リソースを明示的に解放する方法、大抵は `close()` メソッドを提供しています。こうしたオブジェクトは明示的に `close` するよう強く奨めます。操作をする際には、`'try...finally'` 文を使うと便利です。

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。

型はオブジェクトの動作のほとんど全てに影響します。オブジェクトのアイデンティティが重要かどうかでさえ、ある意味では型に左右されます: 変更不能な型では、新たな値を計算するような操作を行うと、実際には同じ型と値を持った既存のオブジェクトへの参照を返すことがあります; 変更可能なオブジェクトではそのような動作は起こりえません。例えば、`'a = 1; b = 1'` とすると、`a` と `b` は値 `1` を持つ同じオブジェクトを参照するときもあるし、そうでないときもあります。これは実装に依存します。しかし、`'c = []; d = []'` とすると、`c` と `d` はそれぞれ二つの異なった、互いに一意な、新たに作成された空のリストを参照することが保証されています。( `'c = d = []'` とすると、`c` と `d` の両方に同じオブジェクトを代入します )

## 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。(C、Java、または実装に使われているその他の言語で書かれた) 拡張モジュールでは、その他に新たな型を定義することができます。将来のバージョンの Python では、型の階層に新たな型 (整数を使って効率的に記憶される有理数型、など) を追加することができるかもしれません。

以下に説明する型のいくつかには、`'特殊属性 (special attribute)'` と題された段落が連ねられています。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。

**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。

**NotImplemented** 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 `NotImplemented` でアクセスされます。数値演算に関するメソッドや拡張比較 (rich comparison) メソッドは、被演算子が該当する演算を行うための実装をもたない場合、この値を返すことがあります。(演算子によっては、インタプリタが関連のある演算を試したり、他の代替操作を行います。) 真値は真 (true) です。

**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組み込み名 `Ellipsis` でアクセスされます。スライス内に `'...'` 構文がある場合に使われます。真値は真 (true) です。

**Numbers** 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。

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。

Python は整数、浮動小数点数、複素数の間で区別を行っています:

**整数型 (integer)** 整数型は、全ての数 (whole numbers) を表す数学的集合内における要素を表現する型です。

以下に三つの整数型を示します:

(通常の) 整数型 (**plain integer**) -2147483648 から 2147483647 までの整数を表現します (基本ワードサイズ: natural word size がより大きなマシンではより大きな定義域になることもあります。より小さくなることはありません。) 演算の結果が定義域を超えた値になった場合、結果は通常長整数で返されます (場合によっては、`OverflowError` が送出されます)。シフト演算やマスク演算のために、整数は 32 ビット以上の 2 の補数で表されたバイナリ表現を持つ (すなわち、4294967296 の異なったビットパターン全てが異なる値を持つ) と仮定されています。

**長整数型 (long integer)** 長整数は無限の定義域を持ち、利用可能な (仮想) メモリサイズの制限のみをうけます。長整数はシフト演算やマスク演算のためにバイナリ表現をもつものと仮定されます。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。

**ブール型 (boolean)** ブール型は、真値 `False` または `True` を表現します。ブール型のオブジェクトは `False` と `True` を表現する二つのオブジェクトだけです。ブール型は整数のサブタイプで、ほとんどの演算コンテキストにおいてブール型値はそれぞれ 0 または 1 のように振舞います。ただし、文字列に変換されたときのみ、それぞれ文字列 `"False"` および `"True"` が返されます。

整数表現に関する規則は、シフト演算やマスク演算において、負の整数も含めて最も有意義な解釈ができるように、かつ通常の整数と長整数との間で定義域を切り替える際にできるだけ混乱しないように決められています。左シフト以外の演算では、演算結果がオーバーフローを起こさずに整数の定義域の値になる場合は、長整数を使った場合でも、被演算子に整数と長整数を混合した場合でも同じ結果になります。

**浮動小数点数型 (floating point number)** この型は計算機レベルで倍精度とされている浮動小数点数を表現します。表現可能な値の範囲やオーバーフローの扱いは、根底にある計算機アーキテクチャ (と C または Java 実装) 次第です。Python は単精度の浮動小数点数をサポートしません; 単精度の数を使う理由は、通常プロセッサやメモリ使用量の節約ですが、こうした節約は Python でオブジェクトを扱う際のオーバーヘッドに比べれば微々たるものにすぎません。従って、わざわざ浮動小数点型を 2 つも定義して Python 言語を難解にする理由はどこにもないのです。

**複素数型 (complex number)** この型は、計算機レベルで倍精度とされている浮動小数点を 2 つ一組にして複素数を表現します。浮動小数点について述べたのと同じ性質が当てはまります。複素数 `z` の実数部および虚数部は、それぞれ読み出し専用属性 `z.real` および `z.imag` で取り出すことができます。

**配列型 (sequence)** この型は、有限の順序集合 (ordered set) を表現します。要素は非負の整数でインデックス化されています。組み込み関数 `len()` を使うと、配列の要素数を返します。配列の長さが  $n$  の場合、インデックスは  $0, 1, \dots, n-1$  からなる集合です。配列  $a$  の要素  $i$  は  $a[i]$  で選択します。

配列はスライス操作 (slice) もサポートしています:  $a[i:j]$  とすると、 $i \leq k < j$  であるインデックス  $k$  をもつ全ての要素を選択します。式表現としてスライスをういた場合、スライスは同じ型をもつ新たな配列を表します。新たな配列内では、インデックス集合が 0 から始まるようにインデックスの値を振りなおします。



配列によっては、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、 $x = i + n*k, n \geq 0$  かつ  $i \leq x < j$  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。

配列は、変更可能なものか、そうでないかで区別されています:

**変更不能な配列 (immutable sequence)** 変更不能な配列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配列型です:

**文字列型 (string)** 文字列の各要素は文字 (character) です。文字型 (character type) は存在しません; 単一の文字は、要素が一つだけの文字列として表現されます。各文字は (少なくとも) 8-bit のバイト列を表現します。組み込み関数 `chr()` および `ord()` を使うと、文字と非負の整数で表されたバイト値の間で変換を行えます。0-127 の値を持つバイト値は、通常同じ ASCII 値をもつ文字を表現していますが、値をどう解釈するかはプログラムにゆだねられています。文字列データ型はまた、例えばファイルから読み出されたデータを記憶するといった用途で、バイト値の配列を表現するために用いられます。

(ネイティブの文字セットが ASCII でないシステムでは、`chr()` や `ord()` が ASCII と EBCDIC との間で対応付けを行っており、文字列間の比較で ASCII 順が守られる限り、文字列の内部表現として EBCDIC を使ってもかまいません。誰か他にもっとましなルールをお持ちですか?)

**Unicode 文字列型** Unicode オブジェクトの各要素は Unicode コード単位です。Unicode コード単位とは、単一の Unicode オブジェクトで、Unicode 序数を表現する 16-bit または 32-bit の値を保持できるものです (この序数の最大値は `sys.maxunicode` で与えられており、コンパイル時に Python がどう設定されているかに依存します)。Unicode オブジェクト内にサロゲートペア (surrogate pair) があってもよく、Python はサロゲートペアを二つの別々の Unicode 要素として報告します。組み込み関数 `unichr()` および `ord()` は、コード単位と非負の整数で表された Unicode 標準 3.0 で定義された Unicode 序数との間で変換を行います。他の文字エンコード形式との相互変換は、Unicode メソッド `encode` および組み込み関数 `unicode()` で行うことができます。

**タプル型 (tuple)** タプルの要素は任意の Python オブジェクトにできます。二つまたはそれ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。

**変更可能な配列型 (mutable sequence)** 変更可能な配列は、作成した後で変更することができます。変更可能な配列では、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。

Python に最初から組み込まれている変更可能な配列型は、今のところ一つだけです:

**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 の配列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。)

拡張モジュール `array` では、別の変更可能な配列型を提供しています。

**マップ型 (mapping)** 任意のインデックス集合でインデックス化された、有限のオブジェクトからなる集合を表します。添字表記 `a[k]` は、`k` でインデッ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、マップ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プ型は、今のところ一つだけです:

**辞書型 (dictionary)** ほとんどどんな値でもインデックスとして使えるような、有限個のオブジェクトからなる集合を表します。キー値 (key) として使えない値は、リストや辞書を含む値や、アイデンティティではなく値でオブジェクトが比較される、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一定であることが必要だからです。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ックスとして使うことができます。辞書は変更可能な型です; 辞書は `{...}` 表記で生成します (5.2.5 節, “辞書表現” を参照してください)。

拡張モジュール `dbm`、`gdbm`、および `bsddb` では、別のマップ型を提供しています。

**呼び出し可能型 (callable type)** 関数呼び出し操作 (5.3.4 節, “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 (7.5 節, “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。

特殊属性: `func_doc` または `__doc__` は関数のドキュメンテーション文字列です。ドキュメンテーションがない場合は `None` になります; `func_name` または `__name__` は関数の名前です; `__module__` は、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; `func_defaults` はデフォルト値を持つ引数に対するデフォルト値が収められたタプルで、デフォルト値を持つ引数がない場合には `None` になります; `func_code` はコンパイルされた関数本体を表現するコードオブジェクトです; `func_globals` は関数のグローバル変数の入った辞書 (への参照) です — この辞書は、関数が定義されているモジュールのグローバルな名前空間を決定します; `func_dict` または `__dict__` には、任意の関数属性をサポートするための名前空間が収められています; `func_closure` は、`None` または関数の個々の自由変数 (引数以外の変数) に対して値を結び付けているセル (cell) 群からなるタプルになります。上記のうち、`func_code`、`func_defaults`、`func_doc/__doc__`、および `func_dict/__dict__` は書き込み可能な属性です; 他の属性は変更できません。関数定義に関するその他の情報は、関数のコードオブジェクトから得られます; 後述の内部型 (internal type) に関する説明を参照してください。

**ユーザ定義メソッド (user-defined method)** ユーザ定義のメソッドオブジェクトは、クラスやクラスインスタンス (あるいは `None`) を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) と結合し (combine) ます。

読み出し専用の特殊属性: `im_self` はクラスインスタンスオブジェクトで、`im_func` は関数オブジェクトです; `im_class` は結合メソッド (bound method) において `im_self` が属しているクラスか、あるいは非結合メソッド (unbound method) において、要求されたメソッドを定義しているクラスです; `__doc__` はメソッドのドキュメンテーション文字列 (`im_func.__doc__` と同じ) です; `__name__` はメソッドの名前 (`im_func.__name__` と同じ) です; `__module__` はメソッドが定義されているモジュールの名前になるか、モジュール名がない場合は `None` になります。2.2 で変更された仕様: メソッドを定義しているクラスを参照するために `im_self` が使われていました

メソッドもまた、根底にある関数オブジェクトの任意の関数属性に (値の設定はできませんが) アクセスできます。

クラスの属性を (おそらくクラスのインスタンスを介して) 取得する際には、その属性がユーザ定義の関数オブジェクト、非結合 (unbound) のユーザ定義メソッドオブジェクト、あるいはクラスメソッドオブジェクトであれば、ユーザ定義メソッドオブジェクトが生成されることがあります。属性がユーザ定義メソッドオブジェクトの場合、属性を取得する対象のオブジェクトが属するクラスがもとのメソッドオブジェクトが定義されているクラスと同じクラスであるか、またはそのサブクラスであれば、新たなメソッドオブジェクトだけが生成されます。それ以外の場合には、もとのメソッドオブジェクトがそのまま使われます。

クラスからユーザ定義関数オブジェクトを取得する方法でユーザ定義メソッドオブジェクトを生成すると、`im_self` 属性は `None` になり、メソッドオブジェクトは非結合 (unbound) であるといえます。クラスのインスタンスからユーザ定義関数オブジェクトを取得する方法でユーザ定義メソッドオブジェクトを生成すると、`im_self` 属性はインスタンスになり、メソッドオブジェクトは結合 (bound) であるといえます。どちらの場合も、新たなメソッドの `im_class` 属性は、メソッドの取得が行われたクラスになり、`im_func` 属性はもとの関数オブジェクトになります。

クラスやインスタンスから他のユーザ定義メソッドオブジェクトを取得する方法でユーザ定義メソッドオブジェクトを生成した場合、その動作は関数オブジェクトの場合と同様ですが、新たなインスタンスの `im_func` 属性はもとのメソッドオブジェクトの属性ではなく、新たなインスタンスの属性になります。

クラスやインスタンスからクラスメソッドオブジェクトを取得する方法でユーザ定義メソッドオブジェクトを生成した場合、`im_self` 属性はクラス自体 (`im_class` 属性と同じ) となり、`im_func` 属性はクラスメソッドの根底にある関数オブジェクトになります。

非結合ユーザ定義メソッドオブジェクトの呼び出しの際には、根底にある関数 (`im_func`) が呼び出されます。このとき、最初の引数は適切なクラス (`im_class`) またはサブクラスのインスタンスでなければならないという制限が課されています。

結合ユーザ定義メソッドオブジェクトの呼び出しの際には、根底にある関数 (`im_func`) が呼び出されます。このとき、クラスインスタンス (`im_self`) が引数の先頭に挿入されます。例えば、関数 `f()` の定義が入ったクラスを `C` とし、`x` を `C` のインスタンスとすると、`x.f(1)` の呼び出しは `C.f(x, 1)` と同じになります。

ユーザ定義メソッドオブジェクトがクラスオブジェクトから導出される際、`im_self` に記憶されている “クラスインスタンス” はクラス自体になります。これは、`x.f(1)` や `C.f(1)` の呼び出しが根底にある関数を `f` としたときの呼び出し `f(C, 1)` と等価になるようにするためです。関数オブジェクトから (結合または非結合の) メソッドオブジェクトへの変換は、クラスやインスタンスから属性を取り出すたびに行われるので注意してください。場合によっては、属性をローカルな変数に代入しておき、その変数を使って関数呼び出しを行うと効果的な最適化になります。また、上記の変換はユーザ定義関数に対してのみ起こるので注意してください; その他の呼び出し可能オブジェクト (および呼び出し可能でない全てのオブジェクト) は、変換を受けずに取り出されます。それから、クラスインスタンスの属性になっているユーザ定義関数は、結合メソッドに変換できないと知っておくことも重要です; 結合メソッドへの変換が行われるのは、関数がクラスの一属性である場合だけです。

ジェネレータ関数 (generator function) `yield` 文 (6.8 節、“`yield` 文” 参照) を使う関数またはメソッドは、ジェネレータ関数 (generator function) と呼ばれます。このような関数は、呼び出された際に、常にイテレータオブジェクトを返します。このイテレータオブジェクトは関数の本体を実行するために用いられます: イテレータの `next()` メソッドを呼び出すと、`yield` 文で値を出力する処理まで関数の実行が行われます。関数が `return` 文を実行するか、関数を最後



まで実行し終わると、`StopIteration` 例外が送出され、イテレータが返す値の集合はそこで終わります。

**組み込み関数 (built-in function)** 組み込み関数オブジェクトは C 関数へのラップです。組み込み関数の例は `len()` や `math.sin()` (`math` は標準の組み込みモジュール) です。引数の数や型は C 関数で決定されています。読み出し専用の特殊属性: `__doc__` は関数のドキュメンテーション文字列です。ドキュメンテーションがない場合は `None` になります; `__name__` は関数の名前です; `__self__` は `None` に設定されています (組み込みメソッドの節も参照してください); `__module__` は、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。

**組み込みメソッド (built-in method)** 実際には組み込み関数を別の形で隠蔽したもので、こちらの場合には C 関数に渡される何らかのオブジェクトを非明示的な外部引数として持っています。組み込みメソッドの例は、`alist` をリストオブジェクトとしたときの `alist.append()` です。この場合には、読み出し専用の属性 `__self__` は `alist` で表されるオブジェクトになります。

**クラス型 (class type)** クラス型、あるいは“新しいクラス型 (new-style class)”や呼び出し可能オブジェクトです。クラス型オブジェクトは通常、そのクラスの新たなインスタンスを生成する際のファクトリクラスとして振舞いますが、`__new__()` をオーバーライドして、バリエーションを持たせることもできます。呼び出しの際に使われた引数は `__new__()` に渡され、さらに典型的な場合では新たなインスタンスを初期化するために `__init__()` に渡されます。

**旧クラス型 (classic class)** (旧) クラスオブジェクトは後で詳しく説明します。クラスオブジェクトが呼び出されると、新たにクラスインスタンス (後述) が生成され、返されます。この操作には、クラスの `__init__()` メソッドの呼び出し (定義されている場合) が含まれています。呼び出しの際に使われた引数は、すべて `__init__()` メソッドに渡されます。`__init__()` メソッドがない場合、クラスは引数なしで呼び出さなければなりません。

**クラスインスタンス (class instance)** クラスインスタンスは後で詳しく説明します。クラスインスタンスはクラスが `__call__()` メソッドを持っている場合にのみ呼び出すことができます; `x(arguments)` とすると、`x.__call__(arguments)` 呼び出しを短く書けます。

**モジュール (module)** モジュールは `import` 文で `import` します (6.12 節、“`import` 文”参照)。モジュールオブジェクトは、辞書オブジェクト (モジュール内で定義されている関数が `func_globals` 属性で参照している辞書です) で実装された名前空間を持っています。属性への参照は、この辞書に対する検索 (lookup) に翻訳されます。例えば、`m.x` は `m.__dict__["x"]` と同じです。モジュールオブジェクトには、モジュールを初期化するために使われるコードオブジェクトは入っていません (一度初期化が終わればもう必要ないからです)。

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。

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。

定義済みの (書き込み可能な) 属性: `__name__` はモジュールの名前です; `__doc__` は関数のドキュメンテーション文字列です。ドキュメンテーションがない場合は `None` になります; モジュールがファイルからロードされた場合、`__file__` は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 モジュールの場合、`__file__` 属性はありません; 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。

**クラス** クラスオブジェクトはクラス定義 (7.6 節、“クラス定義”参照) で生成されます。クラスは辞書で実装された名前空間を持っています。クラス属性への参照は、この辞書に対する検索 (lookup) に翻訳されます。例えば、`C.x` は `C.__dict__["x"]` と同じです。属性がこの検索で見つからない場合、

現在のクラスの基底クラスへと検索を続けます。検索は深さ優先 (depth-first)、かつ基底クラスの挙げられているリスト中の左から右 (left-to-right) の順番で行われます。

クラス (C とします) への属性参照で、要求している属性がユーザ定義関数オブジェクトや、C やその基底クラスに関連付けられている非結合のユーザ定義メソッドオブジェクトである場合、`im_class` 属性が C であるような非結合ユーザ定義メソッドオブジェクトに変換されます。要求している属性がクラスメソッドオブジェクトの場合、`im_class` と `im_self` 属性がどちらも C であるようなユーザ定義メソッドオブジェクトに変換されます。要求している属性が静的メソッドオブジェクトの場合、静的メソッドオブジェクトでラップされたオブジェクトに変換されます。クラスから取り出した属性と実際に `__dict__` に入っているものが異なるような他の場合については、[3.3.2 節](#)を参照してください。

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__` はクラス名です; `__module__` はクラスが定義されているモジュールの名前です; `__dict__` はクラスの名前空間が入った辞書です; `__bases__` は基底クラスの入った (空、あるいは単要素を取りえる) タプルで、基底クラスリストの順番になっています; `__doc__` はクラスのドキュメンテーション文字列です。ドキュメンテーション文字列がない場合には `None` になります。

**クラスインスタンス** クラスインスタンスはクラスオブジェクト (上記参照) を呼び出して生成します。クラスインスタンスは辞書で実装された名前空間を持っており、属性参照の時にはこの辞書が最初に検索されます。辞書内に属性が見つからず、かつインスタンスのクラスに該当する属性名がある場合、検索はクラス属性にまで広がられます。見つかったクラス属性がユーザ定義関数オブジェクトや、インスタンスのクラス (C とします) やその基底クラスに関連付けられている非結合のユーザ定義メソッドオブジェクトの場合、`im_class` 属性が C で `im_self` 属性がインスタンスになっている結合ユーザ定義メソッドオブジェクトに変換されます。静的メソッドやクラスメソッドオブジェクトもまた、C から取り出した場合と同様に変換されます; 上記の“クラス”を参照してください。クラスから取り出した属性と実際に `__dict__` に入っているものが異なるような他の場合については、[3.3.2 節](#)を参照してください。クラス属性が見つからず、かつオブジェクトのクラスが `__getattr__()` メソッドを持っている場合、このメソッドを呼び出して属性名の検索を充足させます。

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、ある特定の名称のメソッドを持っている場合、数値型や配列型、あるいはマップ型のように振舞うことができます。[3.3 節](#)、“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。

**ファイル (file) ファイル オブジェクト** は開かれたファイルを表します。ファイルオブジェクトは組み込み関数 `open()` や、`os.popen()`、`os.fdopen()`、および `socket` オブジェクトの `makefile()` メソッド (その他の拡張モジュールで提供されている関数やメソッド) で生成されます。`sys.stdin`、`sys.stdout` および `sys.stderr` といったオブジェクトは、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よう初期化されます。ファイルオブジェクトに関する完全な記述については、*Python* ライブラリリファレンス を参照してください。

**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、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 (*bytecode*) 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。

読み出し専用の特殊属性: `co_name` は関数名を表します; `co_argcount` は固定引数 (positional argument) の数です; `co_nlocals` は関数を使う (引数を含めた) ローカル変数の数です; `co_varnames` はローカル変数名の入ったタプルです (引数名から始まっています); `co_cellvars` はネストされた関数で参照されているローカル変数の名前が入ったタプルです; `co_freevars` は自由変数の名前が入ったタプルです。 `co_code` はバイトコード列を表現している文字列です; `co_consts` はバイトコードで使われているリテラルの入ったタプルです; `co_names` はバイトコードで使われている名前の入ったタプルです; `co_filename` はバイトコードのコンパイルが行われたファイル名です; `co_firstlineno` は関数の最初の行番号です; `co_lnotab` はバイトコードオフセットから行番号への対応付けをコード化した文字列です (詳細についてはインタプリタのソースコードを参照してください); `co_stacksize` は関数で (ローカル変数の分も含めて) 必要なスタックサイズです; `co_flags` はインタプリタ用の様々なフラグをコード化した整数です。

以下のフラグビットが `co_flags` で定義されています: 0x04 ビットは、関数が `*arguments` 構文を使って任意の数の固定引数を受理できる場合に立てられます; 0x08 ビットは、関数が `**keywords` 構文を使ってキーワード引数を受理できる場合に立てられます; 0x20 ビットは、関数がジェネレータである場合に立てられます。

将来機能 (future feature) 宣言 (`from __future__ import division`) もまた、`co_flags` のビットを立てることで、コードオブジェクトが特定の機能を有効にしてコンパイルされていることを示します: 0x2000 ビットは、関数が将来機能を有効にしてコンパイルされている場合に立てられます; 以前のバージョンの Python では、0x10 および 0x1000 ビットが使われていました。

`co_flags` のその他のビットは将来に内部的に利用するために予約されています。

コードオブジェクトが関数を表現している場合、`co_consts` の最初の要素は関数のドキュメンテーション文字列になります。ドキュメンテーション文字列が定義されていない場合には `None` になります。

**フレーム (frame) オブジェクト** フレームオブジェクトは実行フレーム (execution frame) を表します。実行フレームはトレースバックオブジェクト内に出現します (下記参照)。

読み出し専用の特殊属性: `f_back` は (呼び出し側にとっての) 以前のスタックフレームです。呼び出し側がスタックフレームの最下段である場合には `None` です; `f_code` は現在のフレームで実行しようとしているコードオブジェクトです; `f_locals` はローカル変数を検索するために使われる辞書です; `f_globals` はグローバル変数用です; `f_builtins` は組み込みの (Python 固有の) 名前です; `f_restricted` は、関数が制限つき実行 (restricted execution) モードで実行されているかどうかを示すフラグです; `f_lasti` は厳密な命令コード (コードオブジェクト中のバイトコード文字列へのインデックス) です。

書き込み可能な特殊属性: `f_trace` が `None` でない場合、各ソースコード行の先頭で呼び出される関数になります; `f_exc_type`, `f_exc_value`, `f_exc_traceback` は、現在のフレームでもっとも最近捕捉された例外を表します; `f_lineno` はフレーム中における現在の行番号です — トレース関数 (trace function) 側でこの値に書き込みを行うと、指定した行にジャンプしま

す(最下段の実行フレームにいるときのみ)。デバッガでは、`f_lineno` を書き込むことで、ジャンプ命令(Set Next Statement 命令とも呼ばれます)を実装できます。

**トレースバック (traceback) オブジェクト** トレースバックオブジェクトは例外のスタックトレースを表現します。トレースバックオブジェクトは例外が発生した際に生成されます。例外ハンドラを検索して実行スタックを戻っていく際、戻ったレベル毎に、トレースバックオブジェクトが現在のトレースバックの前に挿入されます。例外ハンドラに入ると、スタックトレースをプログラム側で利用できるようになります(7.4 節“try 文”を参照)。トレースバックは `sys.exc_traceback` として得ることができ、`sys.exc_info()` が返すタプルの三番目の要素としても得られます。インタフェースとしては後者の方が推奨されていますが、これはプログラムがマルチスレッドを使っている場合に正しく動作するからです。プログラムに適切なハンドラがない場合、スタックトレースは(うまく書式化されて)標準エラーストリームに書き出されます; インタプリタが対話的に実行されている場合、`sys.last_traceback` として得ることもできます。

読み出し専用の特殊属性: `tb_next` はスタックトレース内の(例外の発生しているフレームに向かって)次のレベルです。次のレベルが存在しない場合には `None` になります; `tb_frame` は現在のレベルにおける実行フレームを指します; `tb_lineno` は例外の発生した行番号です; `tb_lasti` は厳密な命令コードです。トレースバック内の行番号や最後に実行された命令は、try 文内で例外が発生し、かつ対応する `except` 節や `finally` 節がない場合には、フレームオブジェクト内の行番号とは異なるかもしれません。

**スライス (slice) オブジェクト** スライスオブジェクトは拡張スライス構文 (*extended slice syntax*) が使われた際にスライスを表現するために使われます。拡張スライス構文とは、二つのコロンや、コンマで区切られた複数のスライスや省略符号 (ellipse) を使ったスライスで、例えば `a[i:j:step]`、`a[i:j, k:1]`、あるいは `a[... , i:j]` です。スライスオブジェクトは組み込み関数 `slice()` で生成されます。

読み出し専用の特殊属性: `start` は下境界 (lower bound) です; `stop` は上境界 (upper bound) です; `step` はステップ値 (step value) です; それぞれ省略されている場合には `None` になります。これらの属性は任意の型の値をとることができます。

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:

`indices(self, length)`

このメソッドは単一の整数引数 `length` を取り、`length` 個の要素からなる配列に適用した際にスライスオブジェクトから提供することになる、拡張スライスに関する情報を計算します。このメソッドは三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` のインデクスと、`step` またはスライス間の幅に対応します。インデクス値がないか、範囲外の値である場合、通常のスライスに対して一貫性のあるやりかたで扱われます。2.3 で追加された仕様です。

**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プです。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すことができません。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** クラスメソッドオブジェクトは、静的メソッドオブジェクトに似て、別のオブジェクトを包むラップであり、そのオブジェクトをクラスやクラスインスタンスから取り出す方法を代替します。このようにして取得したクラスメソッドオブジェクトの動作につい



ては、上の“ユーザ定義メソッド (user-defined method)”で説明されています。クラスメソッドオブジェクトは組み込みのコンストラクタ `classmethod()` で生成されます。

### 3.3 特殊メソッド名

特殊な名前をもったメソッドを定義することで、特殊な構文 (算術演算や添え字表記、スライス表記のような) 特定の演算をクラスで実装することができます。これは、個々のクラスが Python 言語で提供されている演算子に対応した独自の振る舞いをできるようにするための、演算子のオーバーロード (*operator overloading*) に対する Python のアプローチです。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、`x[i]` は `x.__getitem__(i)` と等価になります。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場合にこのような演算を行おうとすると例外が送出されます。

組み込み型をエミュレーションするようなクラスを実装する際には、エミュレーションの実装をモデル化しようとしているオブジェクトで意味のある範囲だけにとどめることが重要です。例えば、配列によっては個々の要素の取り出し操作が意味のある操作である一方、スライスの抽出が意味をなさないことがあります。(W3C ドキュメントオブジェクトモデルにおける `NodeList` インタフェースがその一例です。)

#### 3.3.1 基本的なカスタマイズ

`__init__(self, ...)`

インスタンスが生成された際に呼び出されるコンストラクタ (constructor) です。引数はそのクラスのコンストラクタ式に渡した引数になります。基底クラスが `__init__()` メソッドを持っている場合、導出クラスの `__init__()` メソッドでは、例えば `'BaseClass.__init__(self, [args...])'` のように、必要ならば明示的に基底クラスの `__init__()` メソッドを呼び出して、インスタンスの基底クラスに関わる部分が正しく初期化されるようにしなければなりません。コンストラクタには、値を返してはならないという特殊な制限があります; 値を返すようにすると、実行時に `TypeError` の送出を引き起こします。

`__del__(self)`

インスタンスが消滅させられる際に呼び出されます。このメソッドはデストラクタ (destructor) 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、導出クラスの `__del__()` メソッドでは、必要ならば明示的に基底クラスの `__del__()` メソッドを呼び出して、インスタンスの基底クラスに関わる部分が正しく消滅処理されるようにしなければなりません。`__del__()` メソッドでインスタンスに対する新たな参照を作ること、インスタンスの消滅を遅らせることができます (とはいえ、推奨しません!)。このようにすると、新たに作成された参照がその後削除された際にもう一度 `__del__()` メソッドが呼び出されます。インタプリタが終了する際に残っているオブジェクトに対して、`__del__()` メソッドが呼び出される保証はありません。

注意: `'del x'` は直接 `x.__del__()` を呼び出しません — 前者は `x` への参照カウント (reference count) を 1 つ減らし、後者は `x` への参照カウントがゼロになった際にのみ呼び出されます。オブジェクトへの参照カウントがゼロになるのを妨げる可能性のあるよくある状況には、以下のようなものがあります: 複数のオブジェクト間における循環参照 (二重リンクリストや、親子へのポインタを持つツリーデータ構造); 例外を捕捉した関数におけるスタックフレーム上にあるオブジェクトへの参照 (`sys.exc_traceback` に記憶されているトレースバックが、スタックフレームを生き延びさせます); または、対話モードでハンドルされなかった例外を送出したスタックフレーム上にあるオブジェクトへの参照 (`sys.last_traceback` に記憶されているトレースバックが、スタックフレームを生き延びさせます); 最初の状況については、明示的に循環参照を壊すしか解決策はありません; 後者の二つの状況は、`None` を `sys.exc_traceback` や `sys.last_traceback` に入れることで解決で

きます。ごみオブジェクトと化した循環参照は、オプションの循環参照検出機構 (cycle detector) が有効にされている場合 (これはデフォルトの設定です) には検出されますが、検出された循環参照を消去するのは Python レベルで `__del__()` メソッドが定義されていない場合だけです。 `__del__()` メソッドが循環参照検出機構でどのように扱われるか、とりわけ `garbage` 値の記述に関しては、`gc` モジュールのドキュメントを参照してください。

警告: `__del__()` メソッドの呼び出しが起きるのは不安定な状況なので、`__del__()` の実行中に発生した例外は無視され、代わりに `sys.stderr` に警告が出力されます。また、(例えばプログラムの実行終了による) モジュールの削除に伴って `__del__()` が呼び出される際には、`__del__()` メソッドが参照している他のグローバル変数はすでに削除されているかもしれません。この理由から、`__del__()` メソッドでは外部の不変関係を維持する上で絶対最低限必要なことだけをすべきです。バージョン 1.5 からは、単一のアンダースコアで始まるようなグローバル変数は、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ように Python 側で保証しています; これらのアンダースコア付きグローバル変数は、`__del__()` が呼び出された際に、`import` されたモジュールがまだ残っているか確認する上で役に立ちます。

`__repr__(self)`

組み込み関数 `repr()` や、文字列への変換 (逆クオート表記: reverse quote) の際に呼び出され、オブジェクトを表す “公式の (official)” 文字列を計算します。可能な場合には、この値は同じ値を持ったオブジェクトを (適切な環境で) 再生成するために使えるような有効な Python 式に似せるべきです。それが不可能なら、`<...some useful description...>` 形式の文字列を返してください。戻り値は文字列オブジェクトでなければなりません。クラスが `__repr__()` を定義しているが `__str__()` を定義していない場合、そのクラスのインスタンスに対する “非公式の (informal)” 文字列表現が必要なときにも `__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。

`__str__(self)`

組み込み関数 `str()` および `print` 文によって呼び出され、オブジェクトを表す “非公式の (informal)” 文字列を計算します。このメソッドは、有効な Python 式を返さなくても良いという点で、`__repr__()` と異なります: その代わりに、より便利で分かりやすい表現を返すようにしてください。戻り値は文字列オブジェクトでなければなりません。

`__lt__(self, other)`

`__le__(self, other)`

`__eq__(self, other)`

`__ne__(self, other)`

`__gt__(self, other)`

`__ge__(self, other)`

2.1 で追加された仕様です。これらのメソッドは “拡張比較 (rich comparison)” メソッドと呼ばれ、下記の `__cmp__()` に優先して呼び出されます。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;  $x \leq y$  は `x.__le__(y)` を呼び出します;  $x = y$  は `x.__eq__(y)` を呼び出します;  $x \neq y$  および  $x <> y$  は `x.__ne__(y)` を呼び出します;  $x > y$  は `x.__gt__(y)` を呼び出します;  $x \geq y$  は `x.__ge__(y)` を呼び出します。これらのメソッドは任意の値を返すことができますが、比較演算子がブール値のコンテキストで使われた場合、戻り値はブール値として解釈可能でなければなりません。そうでない場合には `TypeError` が送出されます。慣習的には、`False` は偽値、`True` は真値として用いられます。

比較演算子間には、暗黙的な論理関係はありません。すなわち、 $x = y$  が真である場合、暗黙のうちに  $x \neq y$  が偽になるわけではありません。従って、`__eq__` を実装する際、演算子が期待通りに動作するようにするために `__ne__` も定義する必要があります。

これらのメソッドには、(左引数が演算をサポートしないが、右引数はサポートする場合に用いられるような)鏡像となる(引数を入れ替えた)バージョンは存在しません;むしろ、`__lt__()`と`__gt__()`は互いに鏡像であり、`__le__()`と`__ge__()`、および`__eq__()`と`__ne__()`はそれぞれ互いに鏡像です。

拡張比較メソッドの引数には型強制 (coerce) が起こりません。与えられた引数ペアの間で演算が実装されていない場合、拡張比較メソッドは `NotImplemented` を返します。

`__cmp__(self, other)`

拡張比較 (上参照) が定義されていない場合、比較演算によって呼び出されます。`self < other` である場合には負の値、`self == other` ならばゼロ、`self > other` であれば正の値を返さなければなりません。演算 `__cmp__()`、`__eq__()` および `__ne__()` がいずれも定義されていない場合、クラスインスタンスはオブジェクトのアイデンティティ (“アドレス”) で比較されます。自作の比較演算をサポートするオブジェクトや、辞書のキーとして使えるオブジェクトを生成するには、`__hash__()` に関する記述を参照してください。(注意: `__cmp__()` が例外を伝播しないという制限は Python 1.5 から除去されました。)

`__rcmp__(self, other)`

2.1 で変更された仕様: もはやサポートされていません

`__hash__(self)`

辞書演算の際にキーとなるオブジェクトに対して呼び出されたり、組み込み関数 `hash()` から呼び出されたりします。辞書演算におけるハッシュ値として利用できる、32 ビットの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、比較結果が等価であるオブジェクトは同じハッシュ値をもつということです; オブジェクト間で比較を行う際には、オブジェクトの各要素に対するハッシュ値を(排他的論理和をとるなどして) 何らかの方法で混合するよう勧めます。クラスが `__cmp__()` メソッドを定義していない場合、`__hash__()` メソッドも定義してはなりません; クラスが `__cmp__()` または `__eq__()` を定義しているが、`__hash__()` を定義していない場合、インスタンスを辞書のキーとして使うことはできません。クラスが変更可能なオブジェクトを定義しており、`__cmp__()` または `__eq__()` メソッドを実装している場合、`__hash__()` を定義してはなりません。これは、辞書の実装においてハッシュ値が変更不能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、キーが誤ったハッシュバケツ: hash bucket に入っていることになってしまいます)。

`__nonzero__(self)`

真値テストや組み込み演算 `bool()` を実現するために呼び出されます; `False` または `True` か、等価な整数値 0 または 1 を返さなければなりません。このメソッドが定義されていない場合、`__len__()` (下記参照) が定義されていれば呼び出されます。`__len__()` と `__nonzero__()` のどちらもクラスで定義されていない場合、そのクラスのインスタンスはすべて真の値を持つものとみなされます。

`__unicode__(self)`

組み込み関数 `unicode()` を実現するために呼び出されます。Unicode オブジェクトを返さなければなりません。このメソッドが定義されていなければ、文字列への変換が試みられ、その結果がデフォルトの文字エンコードを用いて Unicode に変換されます。

### 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値アクセス (属性値の使用、属性値への代入、`x.name` の削除) の意味をカスタマイズすることができます。

`__getattr__(self, name)`

属性値の検索を行った結果、通常の場所に属性値が見つからなかった場合 (すなわち、`self` のイン

スタンス属性でなく、かつクラスツリーにも見つからなかった場合) 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。

通常メカニズムを介して属性値が見つかった場合、\_\_getattr\_\_() は呼び出されないで注意してください。( \_\_getattr\_\_() と \_\_setattr\_\_() の間は意図的に非対称性にされています。これは \_\_getattr\_\_() および \_\_setattr\_\_() 双方にとっての効率性という理由と、こうしなければ \_\_setattr\_\_() がインスタンスの他の属性値にアクセスする方法がなくなるためです。少なくともインスタンス変数に対しては、値をインスタンスの属性値辞書に挿入しないようにして (代わりに他のオブジェクトに挿入することで) 属性値が完全に制御されているように見せかけられることに注意してください。新形式のクラスで実際に完全な制御を行う方法は、以下の \_\_getattribute\_\_() メソッドを参照してください。

**\_\_setattr\_\_(self, name, value)**

属性値への代入が試みられた際に呼び出されます。このメソッドは通常の代入メカニズム (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値への代入が必要な場合、単に 'self.name = value' としてはなりません — このようにすると、自分自身に対する再帰呼び出しがおきてしまいます。その代わりに、インスタンス属性の辞書に値を挿入してください。例えば、'self.\_\_dict\_\_[name] = value' とします。新しい形式のクラスでは、インスタンス辞書にアクセスするのではなく、基底クラスのメソッドを同じ属性名で呼び出します。例えば、'object.\_\_setattr\_\_(self, name, value)' とします。

**\_\_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。

### 新しい形式のクラスのための別の属性アクセス

以下のメソッドは新しい形式のクラス (new-style class) のみに適用されます。

**\_\_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、\_\_getattribute\_\_() は (明示的に呼び出さない限り) 呼び出されることはありません。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、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し、必要な属性値全てにアクセスしなければなりません。

### デスクリプタ (descriptor) の実装

以下のメソッドは、デスクリプタメソッドを持っているクラス (いわゆるデスクリプタ (descriptor) クラス) のインスタンスが別の新たな形式のクラス、いわゆる オーナ (owner) クラスのクラス辞書に存在する場合にのみ適用されます。以下の例での “属性” とは、属性の名前がオーナクラスの \_\_dict\_\_ に入っているプロパティ (property) を検索するためのキーになっているような属性を指します。デスクリプタ自体は、新しい形式のクラスとしてしか実装できません。

**\_\_get\_\_(self, instance, owner)**

オーナクラスやの属性を取得する (クラス属性へのアクセス) 際や、オーナクラスのインスタンスの属性を取得する (インスタンス属性へのアクセス) 場合に呼び出されます。owner は常にオーナクラスです。一方、instance は属性へのアクセスを仲介するインスタンスが属性が owner を介してアクセス



される場合は `None` になります。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。

```
__set__(self, instance, value)
```

オーナ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。

```
__delete__(self, instance)
```

オーナ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。

## デスクリプタを呼び出す

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えます。

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ことです。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行く、といった具合に連鎖が起こります。

しかしながら、検索対象となる値が、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の属性値である場合、Python はデフォルトの動作をオーバーライドして、デスクリプタメソッドの方を呼び出します。

前後する呼び出し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るかと、どうやってデスクリプタメソッドが呼ばれるかに依存します。デスクリプタは新しい形式のオブジェクトやクラス (`object()` や `type()` をサブクラス化したもの) だけに対して呼び出されるので注意してください。

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)` を行うということです。

束縛 (Instance Binding) 新しい形式のクラスのインスタンスに対する束縛では、`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) 新しい形式のクラスに対する束縛では、`A.x` は呼び出し: `A.__dict__['x'].__get__(None, A)` に変換されます。

スーパー束縛 (Super Binding) `a` が `super` のインスタンスである場合、束縛 `super(B, obj).m()` を行うとまず `A`、続いて `B` に対して `obj.__class__.__mro__` を検索し、次に呼び出し: `A.__dict__['m'].__get__(obj, A)` でデスクリプタを呼び出します。

インスタンス束縛では、デスクリプタ呼び出しの優先順位はどのデスクリプタが定義されているかに依存します。データデスクリプタでは、`__get__()` と `__set__()` を定義します。非データデスクリプタには `__get__()` メソッドしかありません。インスタンス辞書内で属性値が再定義されても、データデスクリプタは常にこの値をオーバーライドします。対照的に、非データデスクリプタの場合には、属性値はインスタンス側でオーバーライドされます。

(`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__`

デフォルトでは、新旧どちらのクラスも、属性の記憶領域として使うための辞書を持っています。この仕様は、ほとんどインスタンス変数を持たないようなオブジェクトの場合には記憶領域の無駄遣いになります。記憶領域の消費量は、大量のインスタンスを生成する際には深刻です。

このデフォルトの設定は、新たな形式のクラス定義において `__slots__` を定義することでオーバーライドできます。`__slots__` 宣言はインスタンス変数の配列を受け取ります。各々のインスタンス上には、各変数の値を記憶するのにちょうど必要な量だけの記憶領域を確保します。各々のインスタンスに対して `__dict__` が生成されることがないので、記憶領域が節約されます。

## `__slots__`

このクラス変数には、文字列、反復可能オブジェクト、あるいはインスタンスが用いる変数名を表す文字列からなる配列を代入することができます。この変数が新しい形式のクラスで定義されている場合、`__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。2.2 で追加された仕様です。

## `__slots__` を利用する際の注意

- `__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__`' を変数名の配列に追加してください。

2.3 で変更された仕様: これまでは、'`__dict__`' を `__slots__` 宣言に追加しても、インスタンス変数名として他にリストされていない新たな属性の代入はできませんでした。

- `__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 reference) はサポートされません。弱参照のサポートが必要ななら、`__slots__` を宣言する際に '`__weakref__`' を変数名の配列に追加してください。2.3 で変更された仕様: これまでは、'`__weakref__`' を `__slots__` 宣言に追加しても、弱参照のサポートを有効にできませんでした。
- `__slots__` は、クラスのレベルで各変数に対するデスクリプタ (3.3.2 を参照) 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。
- 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。
- `__slots__` 宣言が動作するのは、定義が行われたクラスだけに限られています。その結果、サブクラスでは、`__slots__` を定義しない限り `__dict__` を持つことになります。
- `__slots__` は、`long`、`str`、および `tuple` といった、“可変長 (variable-length)” の組み込み型から導出されたクラスでは動作しません。

- `__slots__` には、文字列でない反復可能オブジェクトを代入することができます。辞書型も使うことができます; しかし将来、辞書の各キーに相当する値に何らかの特殊な意味が割り当てられるかもしれません。

### 3.3.3 クラス生成をカスタマイズする

デフォルトでは、新形式クラスは `type()` を使って構築されます。クラス定義が別の名前空間に読み込まれ、クラス名は `type(name, bases, dict)` の結果に結合されます。

クラス定義が読み込まれる際、`__metaclass__` が定義されていれば、`type()` の代わりに `__metaclass__` が指している呼び出し可能オブジェクトが呼び出されます。これによって、

- クラスが生成される前にクラス辞書を変更する
- 他のクラスのインスタンスを返す – 本質的にはファクトリ関数の役割を果たす

といった、クラス生成のプロセスを監視したり置き換えたりするクラスや関数を書くことができます。

`__metaclass__`

この変数は `name`、`bases`、および `dict` を引数として取るような任意の呼び出し可能オブジェクトにできます。クラス生成の際、組み込みの `type()` の代わりに、指定された呼び出しオブジェクトが呼び出されます。2.2 で追加された仕様です。

以下に優先順で並んだ規則によって、適切なメタクラスが決定されます:

- `dict['__metaclass__']` があればそれを使います。
- それ以外の場合で、最低でも一つ基底クラスを持っているなら、基底クラスのメタクラス (`__class__` 属性を探し、なければ基底クラスの型) を使います。
- それ以外の場合で、`__metaclass__` という名前のグローバル変数があれば、それをつかいます。
- それ以外の場合には、旧形式のメタクラス (`types.ClassType`) を使います。

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、ログ記録、インタ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 呼び出し可能オブジェクトをエミュレートする

`__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 `x.__call__(arg1, arg2, ...)` を短く書いたものになります。

### 3.3.5 コンテナをエミュレートする

以下のメソッドを定義して、コンテナオブジェクトを実装することができます。コンテナは通常、(リストやタプルのような) 配列や、(辞書のような) マップ型を指しますが、他のコンテナも同じように表現することができます。最初の一連のメソッドは、配列をエミュレートしたり、マップ型をエミュレートするために使われます; その違いとして、配列の場合には、キーとして許されているのが、配列の長さが  $N$  であるときの  $0 \leq k < N$  なる整数  $k$  か、あるいは要素の範囲を表すスライスオブジェクトでなければならないということです。(後方互換性のため、`__getslice__()` (以下参照) を定義して、拡張されていない

い単純なスライスを扱うようにもできます。) 変更可能な配列では、Python の標準リストオブジェクトのように、メソッド `append()`、`count()`、`index()`、`extend()`、`insert()`、`pop()`、`remove()`、`reverse()`、および `sort()` を提供しなければなりません。マップ型でも、Python の標準辞書オブジェクトのように、`keys()`、`values()`、`items()`、`has_key()`、`get()`、`clear()`、`setdefault()`、`iterkeys()`、`itervalues()`、`iteritems()`、`pop()`、`popitem()`、`copy()`、および `update()` といったメソッドをマップ型で提供するように推奨しています。UserDict モジュールでは、これらのメソッドを `__getitem__()`、`__setitem__()`、`__delitem__()`、および `keys()` といった基本セットから作成する上で役に立つ DictMixin クラスを提供しています。最後に、配列型では以下に述べるメソッド群 `__add__()`、`__radd__()`、`__iadd__()`、`__mul__()`、`__rmul__()`、および `__imul__()` を定義して、(配列間の結合を意味する) 加算操作と (要素の繰り返しを意味する) 乗算操作を実装しなければなりません; `__coerce__()` や、その他の数値演算子を定義してはなりません。マップでも配列でも、`in` 演算子が有効利用できるように `__contains__()` メソッドの定義を推奨します; マップ型では、`in` は `has_key()` と等価でなければなりません; 配列では、配列内の値にわたって検索を行わなければなりません。さらに、マップでも配列でも、コンテナ内にわたる反復操作ができるようにするため、`__iter__()` を実装するように勧めます; マップ型の場合、`__iter__()` は `iterkeys()` と等価でなければなりません; 配列の場合、配列内の値にわたって反復操作を行わなければなりません。

`__len__(self)`

組み込み関数 `len()` を実現するために呼び出されます。オブジェクトの長さを  $\geq 0$  である整数で返さなければなりません。また、オブジェクトが `__nonzero__()` メソッドを定義しておらず、`__len__()` メソッドがゼロを返す場合には、ブール演算コンテキストでは偽であるとみなされます。

`__getitem__(self, key)`

`self[key]` の値評価 (evaluation) を実現するために呼び出されます。配列の場合、キーとして整数とスライスオブジェクトを受理できなければなりません。(配列型をエミュレートする場合) 負のインデックスの解釈は `__getitem__()` メソッド次第となります。`key` が不適切な型であった場合、`TypeError` を送出してもかまいません; (負のインデックス値に対して何らかの解釈を行った上で) `key` が配列のインデックス集合外の値である場合、`IndexError` を送出しなければなりません。注意: `for` ループでは、配列の終端を正しく検出できるようにするために、不正なインデックスに対して `IndexError` が送出されるものと期待しています。

`__setitem__(self, key, value)`

`self[key]` に対する代入を実現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配列の場合だけです。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。

`__delitem__(self, key)`

`self[key]` の削除を実現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、要素を削除できる配列の場合だけです。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。

`__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わたる反復処理ができるような、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。マップの場合、コンテナ内のキーに渡る反復処理でなければならず、かつ `iterkeys()` によって利用できなければなりません。

イテレータオブジェクトでもこのメソッドを実装する必要があります; イテレータの場合、自分自身



を返さなければなりません。イテレータオブジェクトに関するより詳細な情報は、*Python* ライブラリリファレンスの“イテレータ型”を参照してください。

メンバシップテスト演算子 (`in` および `not in`) は通常、配列に渡る反復処理を使って実装されます。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、オブジェクトが配列でなくてもよいようにできます。

`__contains__(self, item)`

メンバシップテスト演算を実現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、キーに対するメンバシップテストを考えなければなりません。

### 3.3.6 配列型エミュレーションで使われるその他のメソッド

以下のオプションとなるメソッドを定義して、配列オブジェクトをより高度にエミュレーションできます。変更不能な配列のメソッドでは、`__getslice__()` が定義できるだけです; 変更可能な配列では三つのメソッド全てを定義できます。

`__getslice__(self, i, j)`

リリース 2.0 以降で撤廃された仕様です。スライスオブジェクトは `__getitem__()` メソッドのパラメタとしてサポートするようになりました。

`self[i:j]` の値評価を実現するために呼び出されます。返されるオブジェクトは `self` と同じ型でなければなりません。スライス表記で `i` や `j` がいない場合には、それぞれゼロや `sys.maxint` に置き換えられるので注意してください。スライスに負のインデックスが用いられた場合、配列の長さがインデックス値に加算されます。インスタンスが `__len__()` メソッドを実装していない場合には、`AttributeError` が送出されます。この計算の結果、インデックス値が負でなくなるという保証はありません。配列の長さよりも大きなインデックス値は修正されません。`__getslice__()` が定義されていない場合、代わりにスライスオブジェクトが生成されて `__getitem__()` に渡されます。

`__setslice__(self, i, j, sequence)`

`self[i:j]` への代入を実現するために呼び出されます。`i` および `j` に関しては、`__getslice__()` と同じ注釈があてはまります。

このメソッドは撤廃されています。`__setslice__()` がいないか、`self[i:j:k]` 形式の拡張スライスの場合には、`__setslice__()` が呼ばれる代わりにスライスオブジェクトが生成され、`__setitem__()` に渡されます。

`__delslice__(self, i, j)`

`self[i:j]` の削除を実現するために呼び出されます。`i` および `j` に関しては、`__getslice__()` と同じ注釈があてはまります。

このメソッドは撤廃されています。`__delslice__()` がいないか、`self[i:j:k]` 形式の拡張スライスの場合には、`__delslice__()` が呼ばれる代わりにスライスオブジェクトが生成され、`__delitem__()` に渡されます。

これらのメソッドは、単一のコロンの使った単一のスライスで、かつスライスメソッドが利用できるときにだけ呼び出されることに注意してください。拡張スライス表記を含んでいるスライス表記や、スライスメソッドがない場合、`__getitem__()`、`__setitem__()`、あるいは `__delitem__()` がスライスオブジェクトを引数として呼び出されます。

以下の例は、プログラムやモジュールを以前のバージョンの *Python* に対して互換性を持たせる方法を示したものです (`__getitem__()`、`__setitem__()`、および `__delitem__()` は引数としてスライスオブジェクトをサポートするものと仮定します):

```

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

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

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

```

`max()` を呼び出していることに注意してください; この呼び出し `__slice__()` メソッド呼び出される前に、負のインデクス値を処理しておくために必要です。負のインデクス値が使われた場合、`__item__()` メソッドは与えられた値をそのまま使いますが、`__slice__()` メソッドは“調理済みの (cooked)”形式になったインデクス値を受け取ります。負のインデクス値が使われると、メソッドを呼び出す前に、常に配列の長さをインデクス値に加算します (加算してもまだ負の値となっていておかまいません); これは、組み込み配列型における慣習的な負のインデクス処理方法で、`__item__()` メソッドでも同様の処理を行うよう期待しています。しかし、ここではすでに負のインデクス値の処理を行っているので、負のインデクスを渡すべきではありません; インデクス値は、`__item__()` メソッドに渡される前に、配列のインデクス集合の境界に制限されていなければなりません。`max(0, i)` を呼び出せば、適切な値を返すので便利です。

### 3.3.7 数値型をエミュレーションする

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。

```

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

```

これらのメソッドは、二項算術演算 (`-`, `*`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) を実現するために呼び出されます。例えば、式  $x+y$  の場合、 $x$  が `__add__()` メソッドをもつクラスのインスタンスであれば、`x.__add__(y)` が呼び出されます。`__divmod__()` メソッドは、`__floordiv__()`

と `__mod__()` を使った場合と等価にならなければなりません; `__truediv__()` (下記参照) と関連づける必要はありません。組み込みの三項演算子バージョンの関数 `pow()` をサポートする場合には、`__pow__()` は、オプションとなる第三の引数を受け取れなくてはなりません。

`__div__(self, other)`

`__truediv__(self, other)`

除算演算 (`/`) は、これらのメソッドで実現されています。 `__truediv__()` は、 `__future__.division` が有効であるときに使われます。それ以外の場合には `__div__()` が使われます。二つのメソッドのうち一方しか定義されていなければ、オブジェクトは他方の演算コンテキストをサポートしなくなります; このとき、`TypeError` が送出されます。

`__radd__(self, other)`

`__rsub__(self, other)`

`__rmul__(self, other)`

`__rdiv__(self, other)`

`__rtruediv__(self, other)`

`__rfloordiv__(self, other)`

`__rmod__(self, other)`

`__rdivmod__(self, other)`

`__rpow__(self, other)`

`__rlshift__(self, other)`

`__rrshift__(self, other)`

`__rand__(self, other)`

`__rxor__(self, other)`

`__ror__(self, other)`

これらのメソッドは二項算術演算 (`+`, `-`, `*`, `/`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) を実現しますが、メソッド呼び出しが行われる被演算子が逆転して (reflected, swapped: 入れ替えられて) います。これらの関数は、左側の被演算子に対応する演算をサポートしていない場合にのみ呼び出されます。例えば、`x-y` において、`y` が `__rsub__()` メソッドを持つクラスのインスタンスである場合、`y.__rsub__(x)` が呼び出されます。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)。

`__iadd__(self, other)`

`__isub__(self, other)`

`__imul__(self, other)`

`__idiv__(self, other)`

`__itruediv__(self, other)`

`__ifloordiv__(self, other)`

`__imod__(self, other)`

`__ipow__(self, other[, modulo])`

`__ilshift__(self, other)`

`__irshift__(self, other)`

`__iand__(self, other)`

`__ixor__(self, other)`

`__ior__(self, other)`

これらのメソッドは、累算算術演算 (augmented arithmetic operations, `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`) を実現するために呼び出されます。これらのメソッドでは、(`self` の値を変更することで) 演算をインプレースで行うよう試み、その結果 (インプレースで演算を行えない場合や、行っていない場合には `self` 自体) を返さなければなりません。特定のメソッドが定義されていない場合、

その累算算術演算は通常のメソッドで代用されます。例えば、 $x+y$  を評価する際、 $x$  が `__iadd__()` メソッドを持つクラスのインスタンスであれば、`x.__iadd__(y)` が呼び出されます。反対に、 $x$  が `__iadd__()` メソッドを持たないクラスのインスタンスであれば、 $x+y$  に基づいて `x.__add__(y)` および `y.__radd__(x)` を考慮します。

```
__neg__(self)
__pos__(self)
__abs__(self)
__invert__(self)
```

単項算術演算 (`-`, `+`, `abs()` および `~`) を実現するために呼び出されます。

```
__complex__(self)
__int__(self)
__long__(self)
__float__(self)
```

組み込み関数 `complex()`, `int()`, `long()`, および `float()` を実現するために呼び出されます。適切な型の値を返さなければなりません。

```
__oct__(self)
__hex__(self)
```

組み込み関数 `oct()` および `hex()` を実現するために呼び出されます。文字列型を返さなければなりません。

```
__coerce__(self, other)
```

“型混合モード (mixed-mode)” での数値間の算術演算を実現するために呼び出されます。 `self` と `other` を共通の数値型に変換して、2 要素のタプルにして返すか、不可能な場合には `None` を返さなければなりません。共通の型が `other` の型になる場合、`None` を返すだけで十分です。この場合、インタプリタはもう一方のオブジェクトを調べて型強制を行おうとするからです (とはいえ、もう一方の値の型が実装上変更できない場合には、ここで `self` を `other` の型に変換しておいた方が便利です)。戻り値に `NotImplemented` を使うのは、`None` を返すのと同じです。

### 3.3.8 型強制規則 (coercion rule)

本節では、型強制 (coercion) に関する規則について記述します。プログラム言語が進化するにつれ、型強制規則について正確に記述するのは難しくなってきました; 従って、あるバージョンのある実装について記述するのは望ましくありません。その代わりに、型強制に関する非公式的なガイドラインを示しておきます。Python 3.0 からは、型強制がサポートされなくなる予定です。

- `%` 演算子の左被演算子が文字列か Unicode オブジェクトの場合、型強制は起きず、文字列としての書式化操作が呼び出されます。
- 型強制演算の定義はもはや推奨されていません。型強制を定義していない混合型 (mixed-mode) 演算は、もとの引数をそのまま演算操作に渡すようになっています。
- 新しい形式のクラス (object から導出されたもの) が、二項演算子に対して `__coerce__()` メソッドを呼び出すことはありません。; `__coerce__()` が呼び出されるのは、組み込み関数 `coerce()` が呼び出されたときだけです。
- 事実上、`NotImplemented` を返す演算子は、全く実装されていないものとして扱われます。
- 以下の説明では、`__op__()` および `__rop__()` は、演算子に相当する一般的なメソッド名を表すために使われます; `__iop__` はインプレース演算子を表します。例えば、演算子 `+` の場合、`__add__()`



および `__radd__()` がそれぞれ左右の被演算子用の二項演算子として使われ、`__iadd__` がインプレース演算用の演算子として使われる、といった具合です。

- オブジェクト  $x$  および  $y$  に対して、まず  $x.__op__(y)$  が試されます。この演算が実装されていないか、`NotImplemented` を返す場合、次に  $y.__rop__(x)$  が試されます。この演算も実装されていないか、`NotImplemented` を返すなら、`TypeError` 例外が送出されます。ただし、以下の例外があるので参照してください:
- 前項に対する例外: 左被演算子が組み込み型や新形式クラスのインスタンスであり、かつ右被演算子が左被演算子と同じクラスか適切なサブクラスのインスタンスである場合、左被演算子の `__op__()` メソッドを試す前に右被演算子の `__rop__()` が試されます。これは、サブクラス側で二項演算子を完全にオーバーライドできるようにするためです。そうしなければ、常に左被演算子の `__op__` メソッドが右被演算子を受理してしまいます: あるクラスのインスタンスが被演算子になるとされている場合、そのサブクラスのインスタンスもまた受理可能だからです。
- 双方の被演算子が型強制を定義している場合、型強制は被演算子の型の `__op__()` や `__rop__()` メソッドが呼び出される前に呼び出され、それより早くなることはありません。型強制の結果、型強制を行うことになったいずれの被演算子とも異なる型が返された場合、返されたオブジェクトの新たな型を使って、この過程が部分的に再度行われます。
- `(+=)` のようなインプレース型の演算子を用いる際、左被演算子が `__iop__()` を実装していれば、`__iop__()` が呼び出され、方強制は一切行われません。演算が `__op__()` かつ/または `__rop__()` に帰着した場合、通常の型強制規則が適用されます。
- $x+y$  において、 $x$  が結合 (concatenation) 演算を実装している配列であれば、配列の結合が実行されます。
- $x*y$  において、一方の演算子が繰り返し (repeat) 演算を実装している配列であり、かつ他方が整数 (`int` または `long`) である場合、配列の繰り返しが実行されます。
- (`__eq__()` ) などのメソッドで実装されている拡張比較は、決して型強制を行いません。( `__cmp__()` で実装されている) 三値比較 (three-way comparison) は、他の二項演算子で行われているのと同じ条件で型強制を受けます。
- 現在の実装では、組み込み数値型 `int`, `long` および `float` は型強制を行いません; 一方、`complex` は型強制を使います。こうした違いは、これらの型をサブクラス化する際に顕在化してきます。そのうち、`complex` 型についても型強制を避けるよう修正されるかもしれません。これらの型は全て、関数 `coerce()` から利用するための `__coerce__()` メソッドを実装しています。



# 実行モデル

## 4.1 名前づけと束縛 (naming and binding)

名前 (*name*) とは、オブジェクトを参照するものを指します。名前への束縛 (*name binding*) 操作を行うと、名前を導入できます。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。

ブロック (*block*) は、Python のプログラムテキストからなる断片で、一つの実行単位となるものです。モジュール、関数本体、そしてクラス定義はブロックです。また、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 (インタプリタに標準入力として与えたり、コマンドラインの第一引数として指定したファイル) は、コードブロックです。スクリプトコマンド (インタプリタのコマンドライン上で `-c` オプションを使って指定したコマンド) もコードブロックです。組み込み関数 `eval()` や `exec` 文に渡した文字列もコードブロックになります。組み込み関数 `input()` から読み取られ、評価される式もまた、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。

スコープ (*scope*) は、ある名前があるブロック内で参照できるかどうかを決めます。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、その名前に対して別の束縛を行っているブロックを除いた、関数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。クラス内で定義された名前のスコープは、クラスのブロック内に制限されます; スコープがメソッドのコードブロックを含むよう拡張されることはありません。

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。

ある名前がブロック内で束縛されている場合、名前はそのブロックにおけるローカル変数 (*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場合、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、ローカル変数でもあるし、グローバル変数でもあります。) ある変数がコードブロック内で使われているが、そのブロックでは定義されていない場合、変数は自由変数 (*free variable*) です。

ある名前の定義がどこにもない場合、`NameError` 例外が送出されます。名前がまだ束縛されていないローカルな変数を参照した場合、`UnboundLocalError` 例外が送出されます。`UnboundLocalError` は、`NameError` のサブクラスです。

名前への束縛は、以下の文構成 (*construct*): 関数の仮引数 (*formal parameter*) 指定、`import` 文、クラスや関数の定義 (定義を行ったブロック中で、クラスや関数名の束縛が行われます)、代入時に、代入対象が識別子である場合、`for` ループのヘッダ、または第二形式の `except` 文ヘッダ、で行われます。“`from...import *`” 形式の `import` 文は、`import` しようとするモジュール内で定義されている名前について、アンダースコ

アから始まっている名前以外の全てを束縛します。この形式は、モジュールレベルでしか使うことができません。

`del` 文で指定された対象は、(`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。外側のスコープで参照されている名前の解放は、不正な操作になります; コンパイラは `SyntaxError` を報告するでしょう。

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。

ある名前束縛操作がコードブロック内のどこかにある場合、ブロック内でその名前を使うと、全て現在のブロックで束縛されている名前を指すものとみなされます。このため、ある名前が束縛される前にブロック内で使われるとエラーを引き起こす可能性があります。

この規則はやや微妙です。Python には宣言文がなく、コードブロックのどこで名前束縛操作を行ってもかまいません。あるコードブロックにおけるローカル変数は、ブロック全体から名前束縛操作が行われている部分を走査して決定します。

`global` 文で指定された名前がブロック内にある場合、その名前は常にトップレベルの名前空間で束縛された名前を参照します。それらの名前はグローバル名前空間、すなわちコードブロックが収められているモジュールの名前空間とモジュール名 `__builtin__` で表される組み込み名前空間、を検索することによって、トップレベルの名前空間で解決されます。グローバル名前空間は、常に最初に検索されます。名前がグローバル名前空間中に見つからない場合、組み込み名前空間が検索されます。`global` 文は、その名前が使われている全ての文に先立って記述されていなければなりません。

あるコードブロックの実行時に関連付けられ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内に入っている名前 `__builtins__` を参照する形になっています; `__builtins__` は辞書がモジュール (後者の場合にはモジュールの辞書が使われます) でなければなりません。通常、`__builtins__` 名前空間は、組み込みモジュール `__builtin__` (注意: ‘s’ なし) のモジュール辞書です。そうでない場合、制限実行 (restricted execution) モードが有効になっています。

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。

グローバル文は、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。

クラス定義は一つの実行文で、名前の使用や定義を行います。クラス定義への参照は、通常の名前解決規則に従います。クラス定義の名前空間は、そのクラスの属性辞書になります。クラスのスコープで定義された名前は、メソッドからは見えません。

#### 4.1.1 動的な機能とのやりとり

自由変数の入った入れ子スコープ (nested scope) を併用すると、Python の文が不正な文になる場合がいくつかあります。

ある変数がスコープの外側から参照された場合、その名前に対する削除操作は不正になります。この場合、コンパイル時にエラーが報告されることになります。

ワイルドカード形式の `import` 文 — `import *` — を関数内で使った場合や、関数が自由変数を含んでいたり、自由変数を伴う入れ子ブロックである場合、コンパイラは `SyntaxError` を送出します。

`exec` が関数内で使われており、関数が自由変数を含んでいたり、自由変数を伴う入れ子ブロックである場合、`exec` に明示的にローカル名前空間を指定しないがぎりコンパイラは `SyntaxError` を送出します。(別の言い方をすれば、`‘exec obj’` は不正になることがあり、`‘exec obj in ns’` はならない、ということです。)

`eval()`、`execfile()`、および `input()` 関数、そして `exec` 文は、名前の解決を行う際に、現在の環境に対して完全にアクセスできるわけではありません。名前が呼び出し側のローカル名前空間やグローバル名前空間から解決されることはあります。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。<sup>1</sup>

`exec` 文と、関数 `eval()` および `execfile()` にはオプションの引数があり、グローバルおよびローカル名前空間をオーバーライドできます。名前空間を一つしか指定しなければ、両方の名前空間として使われます。

## 4.2 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进行处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 (ゼロによる除算など) 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 ... 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。

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場合、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` でない限りバックトレース (*backtrace*) を出力します。

例外は、クラスインスタンスとして識別されます。ある例外にどの `except` 節が一致するかを選択は、オブジェクトのアイデンティティに基づいて行われます。 `except` 節は、同じクラスの例外か、基底クラスの例外しか参照しません。

例外が発行されると、オブジェクト (`None` になることもあります) が例外の 値 (*value*) として渡されます; このオブジェクトが例外ハンドラの選択自体に影響することはありませんが、選択された例外ハンドラに付帯情報として渡されます。例外がクラスの場合、オブジェクトは送出された例外クラスのインスタンスでなければなりません。

警告: 例外に対するメッセージは、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 のバージョンから次のバージョンになるときに、警告なしに変更される可能性があります。したがって、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においては、例外メッセージの内容に依存した記述をすべきではありません。

`try` 文については、[7.4 節](#)、`raise` 文については [6.9 節](#) も参照してください。

---

<sup>1</sup>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。





# 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。

## 5.1 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」と書かれている場合、引数は 3 章の末尾に記載されている型強制規則に基づいて型強制されます。引数がいずれも標準の数値型である場合、以下の型強制が適用されます:

- 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;
- それ以外の場合で、片方の引数が浮動小数点数であれば、他方は浮動小数点型に変換されます;
- それ以外の場合で、片方の引数が長整数型であれば、他方は長整数型に変換されます;
- それ以外の場合で、両方の引数が通常の整数型であれば、変換の必要はありません。

特定の演算子 (文字列を左引数とする '%' 演算子など) では、さらに別の規則が適用されます。拡張をおこなうことで、個々の演算子に対する型強制を定義できます。

## 5.2 アトム、原子的要素 (atom)

アトム (原子的要素: atom) は、式を構成する基本単位です。もっとも単純なアトムは、識別子またはリテラルになります。逆クオートや丸括弧、波括弧、または角括弧で囲われた形式 (form) もまた、文法的にはアトムに分類されます。アトムの構文定義は以下のようになります:

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

### 5.2.1 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。名前づけや束縛については、4.1 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、名前アトム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、アトムを評価しようとすると `NameError` 例外を送出します。

プライベートな名前の難号化 (**mangling**): クラス定義内にテキストの形で書かれた識別子で、二つ以上のアンダースコアから始まり、末尾が二つ以上のアンダースコアになっていないものは、そのクラスの プライベートな名前 (*private name*) とみなされます。プライベートな名前は、コードが生成される前に、より長い形式の名前に変換されます。この変換では、クラス名の先頭にあるアンダースコアを全てはぎとり、先頭にアンダースコアを一つ挿入して、名前の前に付加します。例えば、クラス `Ham` 内の識別子 `__spam` は、`_Ham__spam` に変換されます。変換は識別子が使われている構文的コンテキストとは独立しています。変換された名前が非常に長い (255 文字以上) の場合には、実装によっては名前の切り詰めが起きるかもしれませんが、クラス名がアンダースコアだけから成り立つ場合には、変換は行われません。

## 5.2.2 リテラル

Python では、文字列リテラルと、様々な数値リテラルをサポートしています:

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

リテラルを評価すると、指定した型 (文字列、整数、長整数、浮動小数点数、複素数) の指定した値を持つオブジェクトになります。浮動小数点や虚数 (複素数) リテラルの場合、値は近似値になる場合があります。詳しくは [2.4](#) を参照してください。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。

## 5.2.3 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 ::= "(" [expression_list] ")"
```

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見られると、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う。

## 5.2.4 リスト表現

リスト表現は、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:

```

test          ::= and_test ( "or" and_test ) * | lambda_form
testlist      ::= test ( "," test ) * [ "," ]
list_display  ::= "[" [listmaker] "]"
listmaker     ::= expression ( list_for | ( "," expression ) * [ "," ] )
list_iter     ::= list_for | list_if
list_for      ::= "for" expression_list "in" testlist [list_iter]
list_if       ::= "if" test [list_iter]

```

リスト表現は、新に作成されたリストオブジェクトを表します。新たなリストの内容は、式のリストを与えるか、リストの内包表記 (list comprehension) で指定します。カンマで区切られた式のリストを与えた場合、リストの各要素は左から右へと順に評価され、評価された順番にリスト内に配置されます。リストの内包表記を与える場合、内包表記はまず単一の式、続いて少なくとも一つの `for` 節、続いてゼロ個以上の `for` 節か、`if` 節になります。この場合、新たに作成されるリストの各要素は、各々の `for` や `if` 節を左から右の順にネストしたブロックとみなして実行し、ネストの最内ブロックに到達する度に式を評価した値となります。

### 5.2.5 辞書表現

辞書表現は、波括弧で囲われた、キーと値のペアからなる系列です。系列は空の系列であってもかまいません:

```

dict_display  ::= "{" [key_datum_list] "}"
key_datum_list ::= key_datum ( "," key_datum ) * [ "," ]
key_datum     ::= expression ":" expression

```

辞書表現は、新たな辞書オブジェクトを表します。

キー/データのペアは、左から右へと評価され、その結果が辞書の各エントリを決定します: 各キーオブジェクトは、対応するデータを辞書に記憶するためのキーとして用いられます。

キーの値として使える型に関する制限は、[3.2 節](#)ですすでに列挙しています。(一言でいうと、キーは変更可能なオブジェクトを全て排除したハッシュ可能な型でなければなりません。) 重複するキー間で衝突が起きても、衝突が検出されることはありません; あるキーに対して、最後に渡されたデータ (プログラムテキスト上では、辞書表記の最も右側値となるもの) が使われます。

### 5.2.6 文字列変換

文字列変換は、逆クオート (reverse quote, 別名バッククオート: backward quote) で囲われた式のリストです:

```

string_conversion ::= "`" expression_list "`"

```

文字列変換は、逆クオート内の式リストを評価して、評価結果のオブジェクトを各オブジェクトの型特有の規則に従って文字列に変換します。

オブジェクトが文字列、数値、`None` か、それらの型のオブジェクトのみを含むタプル、リストまたは辞書の場合、評価結果の文字列は有効な Python 式となり、組み込み関数 `eval()` に渡した場合に同じ値となります (浮動小数点が含まれている場合には近似値の場合もあります)。

(特に、文字列を変換すると、値を安全に出力するために文字列の両側にクオートが付けられ、“変 (funny) な” 文字はエスケープシーケンスに変換されます。)

再帰的な構造をもつオブジェクト (例えば自分自身を直接または間接的に含むリストや辞書) では、`'...'` を使って再帰的参照であることが示され、オブジェクトの評価結果は `eval()` に渡しても等価な値を得ることができません (`SyntaxError` が送出されます)。

組み込み関数 `repr()` は、括弧内の引数に対して、逆クオート表記で囲われた中身と全く同じ変換を実

行します。組み込み関数 `str()` は似たような動作をしますが、もっとユーザフレンドリな変換になります。

## 5.3 一次語 (primary)

一次語は、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
```

### 5.3.1 属性参照

属性参照は、一次語の後ろにピリオドと名前を連ねたものです:

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

一次語の値評価結果は、例えばモジュール、リスト、インスタンスといった、属性参照をサポートする型でなければなりません。オブジェクトは次に、指定した名前が識別子名となっているような属性を生成するよう問い合わせられます。問い合わせた属性が得られない場合、例外 `AttributeError` が送出されます。それ以外の場合、オブジェクトは属性オブジェクトの型と値を決定し、生成して返します。同じ属性参照を複数回評価したとき、互いに異なる属性オブジェクトになることがあります。

### 5.3.2 添字表記 (subscription)

添字表記は、配列 (文字列、タプルまたはリスト) やマップ (辞書) オブジェクトから、要素を一つ選択します:

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

一次語の値評価結果は、配列型かマップ型のオブジェクトでなければなりません。

一次語がマップであれば、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プ内の値 (value) を選択します。(式リストの要素が単独である場合を除き、式リストはタプルでなければなりません。)

一次語が配列の場合、式 (リスト) の値評価結果は (通常の) 整数でなければなりません。値が負の場合、配列の長さが加算されます (`x[-1]` が `x` の最後の要素を指すことになります)。加算結果は配列内の要素数よりも小さな非負の整数とならなければなりません。添字表記は、添字と同じ配列中の (ゼロから数えた) インデックスを持つ要素を選択します。

文字列型の要素は文字 (character) 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。

### 5.3.3 スライス表記 (slicing)

スライス表記は配列オブジェクト (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ようになります:

```

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

上記の形式的な構文法にはあいまいさがあります: 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ということです。この場合には、(スライスリストの評価結果が、適切なスライスや省略表記 (ellipsis) にならない場合)、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の方が高い優先順位を持つように定義することで、構文法をより難解にすることなくあいまいさを取り除いています。同様に、スライスリストが厳密に一つだけの短いスライスで、末尾にカンマが続いていない場合、拡張スライスとしての解釈より、単純なスライスとしての解釈が優先されます。

単純なスライスに対する意味付けは以下ようになります。一次語の値評価結果は、配列型のオブジェクトでなければなりません。下境界および上境界を表す式がある場合、それらの値評価結果は整数でなくてはなりません; デフォルトの値は、それぞれゼロと `sys.maxint` です。どちらかの境界値が負である場合、配列の長さが加算されます。こうして、スライスは  $i$  および  $j$  をそれぞれ指定した下境界、上境界として、インデクス  $k$  が  $i \leq k < j$  となる全ての要素を選択します。選択の結果、空の配列になることもあります。  $i$  や  $j$  が有効なインデクス範囲の外側にある場合でも、エラーにはなりません (範囲外の要素は存在しないので、選択されないだけです)。

拡張スライスに対する意味付けは、以下ようになります。一次語の値評価結果は、辞書型のオブジェクトでなければなりません。また、辞書は以下に述べるようにしてスライスリストから生成されたキーによってインデクス指定できなければなりません。スライスリストに少なくとも一つのカンマが含まれている場合、キーは各スライス要素を値変換したものからなるタプルになります; それ以外の場合、単一のスライス要素自体を値変換したものがキーになります。一個の式でできたスライス要素の変換は、その式になります。省略表記スライス要素の変換は、組み込みの `Ellipsis` オブジェクトになります。適切なスライスの変換は、スライスオブジェクト (3.2 参照) で、`start`, `stop` および `step` 属性は、それぞれ指定した下境界、上境界、およびとび幅 (stride) になります。式がない場合には、`None` に置き換えられます。

### 5.3.4 呼び出し (call)

呼び出し (call) は、呼び出し可能オブジェクト (callable object, 例えば関数など) を、引数列とともに呼び出します。引数列は空の配列でもかまいません:



```

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

positional_arguments ::= expression (", " expression)*
keyword_arguments    ::= keyword_item (", " keyword_item)*
keyword_item         ::= identifier "=" expression

```

固定引数と引数リストの末尾にカンマがあってもよく、構文の意味付けに影響を及ぼすことはありません。

一次語の値評価結果は、呼び出し可能オブジェクトでなければなりません (ユーザ定義関数、組み込み関数、組み込みオブジェクトのメソッド、クラスオブジェクト、クラスインスタンスのメソッド、そして特定のクラスインスタンス自体が呼び出し可能です; 拡張によって、その他の呼び出し可能オブジェクト型を定義することができます)。引数式は全て、呼び出しを試みる前に値評価されます。仮引数 (formal parameter) リストの構文については、[7.5](#) を参照してください。

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固定引数 (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 個の固定引数がある場合、固定引数は先頭の 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パラメタ名と同じなら、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、引数値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。

仮引数スロットの数よりも多くの固定引数がある場合、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固定引数が入ったタプル (もしくは、余分な固定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。

関数呼び出しの際に `*expression` 構文が使われる場合、`expression` の値評価結果は配列でなくてはなりません。この配列の要素は、追加の固定引数のように扱われます; すなわち、固定引数  $x_1, \dots, x_N$  と、 $y_1, \dots, y_M$  になる配列 `expression` を使った場合、 $M+N$  個の固定引数  $x_1, \dots, x_N, y_1, \dots, y_M$  を使った呼び出しと同じになります。

上記の仕様による結果として、`*expression` 構文はたとえキーワード引数 以降に あっても、キーワード引数以前に (`**expression` 引数があればさらにその後— 下記参照) 処理されます。従って:



```

>>> 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 ?
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2

```

となります。

キーワード引数と `*expression` 構文を同じ呼び出しに使うことはあまりないので、実質的には上記のような混乱が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` の値評価結果は辞書 (またはそのサブクラス) でなければなりません。辞書の内容は追加のキーワード引数として扱われます。明示的なキーワード引数が `expression` 内のキーワードと重複した場合には、`TypeError` 例外が送出されます。

`*identifier` や `**identifier` 構文を使った仮引数は、固定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。`(sublist)` 構文を使った仮引数は、キーワード引数名には使えません; `sublist` は、リスト全体が一つの無名の引数スロットに対応しており、`sublist` 中の引数は、他の全てのパラメタに対する処理が終わった後に、通常のタプル形式の代入規則を使ってスロットに入れられます。

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。

呼び出し可能オブジェクトが。。。

ユーザ定義関数のとき: 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 [7.5](#) 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。

組み込み関数や組み込みメソッドのとき: 結果はインタプリタに依存します; 組み込み関数や組み込みメソッドの詳細は、*Python* ライブラリリファレンス を参照してください。

クラスオブジェクトのとき: 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のとき: 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のとき: クラスで `__call__()` メソッドが定義されていなければなりません; `__call__()` メソッドが呼び出された場合と同じ効果をもたらします。

## 5.4 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があります; 一方、右側にある単項演算子よりは低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 ::= 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 (この演算規則は、被演算子の評価順序を縛る規則ではありません)。

べき乗演算子は、二つの引数で呼び出される組み込み関数 `pow()` と同じ意味付けを持っています。引数はまず共通の型に変換されます。結果の型は、型強制後の引数の型になります。

引数型を混合すると、二項算術演算における型強制規則が適用されます。整数や長整数の被演算子の場合、第二引数が負でない限り、結果は (型強制後の) 被演算子と同じになります; 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。(上述の仕様のうち、最後のものは Python 2.2 で追加されました。Python 2.1 以前では、双方の引数が整数型で、第二引数が負の場合、例外が送出されていました。)

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すると `ValueError` になります。

## 5.5 単項算術演算 (unary arithmetic operation)

全ての単項算術演算 (およびビット単位演算子) は、同じ優先順位を持っています:

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

単項演算子 `-` (マイナス) は、引数となる数値の符号を反転 (invert) します。

単項演算子 `+` (プラス) は、数値引数を変更しません。

単項演算子 `~` (逆転) は、整数または長整数の引数をビット単位反転 (bit-wise invert) します。`x` のビット単位反転は、`-(x+1)` として定義されています。この演算子は整数にのみ適用されます。

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。

## 5.6 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 (multiplicative) 演算子と加算的 (additive) 演算子しかありません:

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

`*` (乗算: multiplication) 演算は、引数間の積になります。引数の組は、双方ともに数値型であるか、片方が整数 (通常の整数または長整数) 型で他方が配列型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、配列の繰り返し操作が行われます。繰り返し数を負にすると、空の配列になります。

`/` (除算: division) および `//` (切り捨て除算: floor division) は、引数間の商になります。数値引数はまず共通の型に変換されます。整数または長整数の除算結果は、同じ型の整数になります; この場合、結果は数学的な除算に関数 `'floor'` を適用したのになります。ゼロによる除算を行うと `ZeroDivisionError` 例外を送出します。

`%` (モジュロ: modulo) 演算は、第一引数を第二引数で除算したときの剰余になります。数値引数はまず共通の型に変換されます。右引数値がゼロの場合には、`ZeroDivisionError` 例外が送出されます。引数値は浮動小数点でもよく。例えば `3.14%0.7` は 0.34 になります (`3.14` は `4*0.7 + 0.34` だからです)。モジュロ演算子は常に第二引数と同じ符号 (またはゼロ) の結果になります; モジュロ演算の結果の絶対値は、常に第二引数の絶対値よりも小さくなります。<sup>1</sup>

<sup>1</sup> `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

整数による除算演算やモジュロ演算は、恒等式:  $x == (x/y)*y + (x\%y)$  と関係しています。整数除算やモジュロはまた、組み込み関数 `divmod()`: `divmod(x, y) == (x/y, x%y)` と関係しています。これらの恒等関係は浮動小数点の場合には維持されません;  $x/y$  が `floor(x/y)` や `floor(x/y) - 1` に置き換えられた場合、これらの恒等式は近似性を維持します。<sup>2</sup>

リリース 2.3 以降で撤廃された仕様です。切り捨て除算演算子、モジュロ演算子、および `divmod()` 関数は、複素数に対してはもはや定義されていません。目的に合うならば、代わりに `abs()` を使って浮動小数点に変換してください。

+ (加算) 演算は、引数を加算した値を返します。引数は双方とも数値型か、双方とも同じ型の配列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、配列は結合 (concatenate) されます。

- (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。

## 5.7 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
```

シフトの演算子は整数または長整数を引数にとります。引数は共通の型に変換されます。シフト演算では、最初の引数を二つ目の引数に応じたビット数だけ、左または右にビットシフトします。

$n$  ビットの右シフトは、`pow(2, n)` による除算として定義されています。 $n$  ビットの左シフトは、`pow(2, n)` による乗算として定義されています; 整数の場合、桁あふれ (overflow) のチェックはされないので、演算によって末端のビットは捨てられます。また、結果の絶対値が `pow(2, 31)` よりも小さくない場合には、符号の反転が起こります。負のビット数でシフトを行うと、`ValueError` 例外を送出します。

## 5.8 ビット単位演算の二項演算 (binary bit-wise 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
```

& 演算子は、引数間でビット単位の AND をとった値になります。引数は整数または長整数でなければなりません。引数は共通の型に変換されます。

^ 演算子は、引数間でビット単位の XOR (排他的 OR) をとった値になります。引数は整数または長整数でなければなりません。引数は共通の型に変換されます。

| 演算子は、引数間でビット単位の OR (非排他的 OR) をとった値になります。引数は整数または長整数でなければなりません。引数は共通の型に変換されます。

## 5.9 比較 (comparison)

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:

には厳密に `1e100` と等価です。math モジュールの関数 `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。

<sup>2</sup>  $x$  が  $y$  の整数倍に非常に近い場合、丸め誤差によって `floor(x/y)` は  $(x-x\%y)/y$  よりも大きな値に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。

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

比較演算の結果はブール値: True または False になります。

比較はいくらでも連鎖することができます。例えば  $x < y \leq z$  は  $x < y$  and  $y \leq z$  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、 $a, b, c, \dots, y, z$  が式で、 $opa, opb, \dots, opy$  が比較演算子である場合、 $a opa b opb c \dots y opy z$  は  $a opa b$  and  $b opb c$  and  $\dots y opy z$  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。

$a opa b opb c$  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。

$<>$  と  $!=$  の二つの形式は等価です; C との整合性を持たせるためには、 $!=$  を推奨します; 以下で  $!=$  について触れている部分では、 $<>$  を使うこともできます。 $<>$  のような書き方は、現在では古い書き方とみなされています。

演算子  $<, >, ==, >=, <=$ , および  $!=$  は、二つのオブジェクト間の値を比較します。オブジェクトは同じ型である必要はありません。双方のオブジェクトが数値であれば、共通型への変換が行われます。それ以外の場合、異なる型のオブジェクトは常に不等であるとみなされ、一貫してはいるが規定されていない方法で並べられます。

(このような比較演算の変則的な定義は、ソートのような操作や、 $in$  および  $not in$  といった演算子の定義を単純化するためのものです。将来、異なる型のオブジェクト間における比較規則は変更されるかもしれません。)

同じ型のオブジェクト間における比較は、型によって異なります:

- 数値間の比較では、算術的な比較が行われます。
- 文字列間の比較では、各文字に対する等価な数値型 (組み込み関数 `ord()` の結果) を使って辞書的な (lexicographically) 比較が行われます。Unicode および 8 ビット文字列は、この動作に関しては完全に互換です。
- タプルやリスト間の比較では、対応する各要素の比較結果を使って辞書的な比較が行われます。このため、二つの配列を等価にするためには、各要素が完全に等価でなくてはならず、配列は同じ型で同じ長さをもっていなければなりません。

二つの配列が等価でない場合、異なる値を持つ最初の要素間での比較に従った順序関係になります。例えば、`cmp([1,2,x], [1,2,y])` は `cmp(x,y)` と等しい結果を返します。片方の要素に対応する要素が他方ない場合、より短い配列が前に並びます (例えば、 $[1,2] < [1,2,3]$  となります)。

- マップ (辞書) 間の比較では、(key, value) からなるリストをソートしたものが等しい場合に等価になります。<sup>3</sup>等価性評価以外の結果は一貫したやりかたで解決されるか、定義されないかのいずれかです。<sup>4</sup>
- その他のほとんどの型の比較では、同じオブジェクトでないかぎり等価にはなりません。あるオブジェクトの他のオブジェクトに対する大小関係は任意に決定され、一つのプログラムの実行中は一貫したものとなります。

<sup>3</sup>実装では、この演算をリストを構築したりソートしたりすることなく効率的に行います。

<sup>4</sup>Python の初期のバージョンでは、ソートされた (key, value) のリストに対して辞書的な比較を行っていましたが、これは等価性の計算のようなよくある操作を実現するには非常にコストの高い操作でした。もっと以前のバージョンの Python では、辞書はアイデンティティだけで比較されていました。しかしこの仕様は、 $\{\}$  との比較によって辞書が空であるか確かめられると期待していた人々を混乱させていました。



演算子 `in` および `not in` は、集合内の要素であるかどうか (メンバシップ、membership) を調べます。 `x in s` は、 `x` が集合 `s` のメンバである場合には真となり、それ以外の場合には偽となります。 `x not in s` は `x in s` の否定 (negation) を返します。集合メンバシップテストは、伝統的には配列型に限定されてきました; すなわち、あるオブジェクトがある集合のメンバとなるのは、集合が配列型であり、配列がオブジェクトと等価な要素を含む場合でした。しかしながら、現在ではオブジェクトが配列でなくてもメンバシップテストをサポートしています。特に、辞書型では、 `key in dict` と書くことで、うまい具合にメンバシップテストをサポートしています; 他のマップ型もこれに倣っているかもしれません。

リストやタプル型については、 `x in y` は `x == y[i]` となるようなインデクス `i` が存在するとき、かつそのときに限り真になります。

Unicode 文字列または文字列型については、 `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真になります。この演算と等価なテストは `y.find(x) != -1` です。 `x` および `y` は同じ型である必要はないので注意してください。すなわち、 `u'ab' in 'abc'` は `True` を返すことになります。空文字列は、他のどんな文字列に対しても常に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。2.3 で変更された仕様: 以前は、 `x` は長さ 1 の文字列型でなければなりませんでした

`__contains__()` メソッドの定義されたユーザ定義クラスでは、 `x in y` が真となるのは `y.__contains__(x)` が真となるとき、かつそのときに限ります。

`__contains__()` は定義していないが `__getitem__()` は定義しているようなユーザ定義クラスでは、 `x in y` は `x == y[i]` となるような非負の整数インデクス `i` が存在するとき、かつそのときに限り真となります。インデクス `i` が負である場合に `IndexError` 例外が送出されることはありません。(別の何らかの例外が送出された場合、例外は `in` から送出されたかのようになります)。

演算子 `not in` は、 `in` の真値に対する逆転として定義されています。

演算子 `is` および `is not` は、オブジェクトのアイデンティティ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。 `x is not y` は、 `is` の真値を逆転したものになります。

## 5.10 ブール演算 (boolean operation)

ブール演算は、全ての Python 演算子の中で、最も低い優先順位になっています:

```
expression ::= or_test | lambda_form
or_test     ::= and_test | or_test "or" and_test
and_test    ::= not_test | and_test "and" not_test
not_test    ::= comparison | "not" not_test
lambda_form ::= "lambda" [parameter_list]: expression
```

ブール演算のコンテキストや、式が制御フロー文中で使われる最には、以下の値: `None`、すべての数値型におけるゼロ、空の配列 (文字列、タプル、およびリスト)、空のマップ型 (辞書)、は偽 (`false`) であると解釈されます。それ以外の値は真 (`true`) であると解釈されます。

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` の値を評価し、その結果を返します。

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` の値を評価し、その結果を返します。

(`and` も `not` も、返す値を `False` や `True` に制限するのではなく、最後に評価した引数の値を返すので注意してください。この仕様は、例えば `s` を文字列として、 `s` が空文字列の場合にデフォルトの値に置き換えるような場合に、 `s or 'foo'` と書くとき期待通りの値になるために便利があります。 `not` は、

式の値でなく独自に値を作成して返すので、引数と同じ型の値を返すような処理に煩わされることはありません。例えば、`not 'foo'` は、`"` ではなく `False` になります)

## 5.11 ラムダ (lambda)

ラムダ形式 (lambda form, ラムダ式 (lambda expression)) は、構文法的には式と同じ位置付けになります。ラムダは、無名関数を作成できる省略記法です; 式 `lambda arguments: expression` は、関数オブジェクトになります。ラムダが表す無名オブジェクトは、以下のコード

```
def name(arguments):  
    return expression
```

で定義された関数と同様に動作します。

引数リストの構文法については、[7.5 節](#)を参照してください。ラムダ形式で作成された関数は、実行文 (statement) を含むことができないので注意してください。

## 5.12 式のリスト

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

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、リスト中の式の数に等しくなります。リスト中の式は左から右へと順に評価されます。

単一要素のタプル (別名单集合 (*singleton*)) を作りたければ、末尾にカンマが必要です。単一の式だけで、末尾にカンマをつけない場合には、タプルではなくその式の値になります (空のタプルを作りたいなら、中身が空の丸括弧ペア: `()` を使います。)

## 5.13 評価順序

Python は、式を左から右へと順に評価してゆきます。ただし、代入式を評価する最には、代入演算子の右側項が左側項よりも先に評価されるので注意してください。

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)  
func(expr1, expr2, *expr3, **expr4)  
expr3, expr4 = expr1, expr2
```

## 5.14 まとめ

以下の表は、Python における演算子を、優先順位 の最も低い (結合度が最も低い) ものから最も高い (結合度が最も高い) ものの順に並べたものです。同じボックス内に示された演算子は同じ優先順位を持ちます。演算子の文法が示されていないかぎり、演算子は全て二項演算子です。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 (値のテストを含む比較演算子を除きます。比較演算子は、左から右に連鎖し



ます — 5.9 を参照してください。また、べき乗演算子も除きます。べき乗演算子は右から左にグループ化されます)。

演算子	説明
<code>lambda</code>	ラムダ式
<code>or</code>	ブール演算 OR
<code>and</code>	ブール演算 AND
<code>not x</code>	ブール演算 NOT
<code>in, not in</code> <code>is, is not</code> <code>&lt;, &lt;=, &gt;, &gt;=, &lt;&gt;, !=, ==</code>	メンバシップテスト アイデンティティテスト 比較
<code> </code>	ビット単位 OR
<code>^</code>	ビット単位 XOR
<code>&amp;</code>	ビット単位 AND
<code>&lt;&lt;, &gt;&gt;</code>	シフト演算
<code>+, -</code>	加算および減算
<code>*, /, %</code>	乗算、除算、剰余
<code>+x, -x</code> <code>~x</code>	正符号、負符号 ビット単位 NOT
<code>**</code>	べき乗
<code>x.attribute</code> <code>x[index]</code> <code>x[index:index]</code> <code>f(arguments...)</code>	属性参照 添字指定 スライス操作 関数呼び出し
<code>(expressions...)</code> <code>[expressions...]</code> <code>{key:datum...}</code> <code>`expressions...`</code>	式結合またはタプル表現 リスト表現 辞書表現 文字列への型変換



## 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
            | pass_stmt
            | del_stmt
            | print_stmt
            | return_stmt
            | yield_stmt
            | raise_stmt
            | break_stmt
            | continue_stmt
            | import_stmt
            | global_stmt
            | exec_stmt
```

### 6.1 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; 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 ::= expression_list
```

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` でない場合、値を組み込み関数 `repr()` で文字列に変換して、その結果のみからなる一行を標準出力に書き出します (6.6 節参照)。(None になる式文の値は書き出されないの、プロシジャ呼び出しを行っても出力は得られません。)

### 6.2 Assert 文 (assert statement)

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、

```
if __debug__:
    if not expression1: raise AssertionError, expression2
```

と等価です。

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__` は通常は 1 であり、インタプリタに (コマンドラインオプション `-O` で) 最適化を要求すると 0 になります。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` 文に対するコードを全く出力しません。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; メッセージ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。

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

(末尾の三つのシンボルの構文については 5.3 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、得られた単一の結果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 (3.2 節を参照してください)。

ターゲットリストへ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。

- ターゲットリストが単一のターゲットからなる場合: オブジェクトはそのターゲットに代入されます。
- ターゲットリストが、カンマで区切られた複数のターゲットからなるリストの場合: オブジェクトはターゲットリスト中のターゲット数と同じ数の要素からなる配列でなければならない、その各要素は左

から右へと対応するターゲットに代入されます。(これは Python 1.5 で緩和された規則です; 以前のバージョンでは、代入するオブジェクトはタプルでなければなりませんでした。文字列も配列なので、今では `'a', b = "xy"` のような代入は文字列が正しい長さを持つ限り正規の操作になります。)

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` 文に書かれていない場合: 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。
  - それ以外の場合: 名前は現在のグローバル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。

- ターゲットが丸括弧や角括弧で囲われたターゲットリストの場合: オブジェクトはターゲットリスト中のターゲット数と同じ数の要素からなる配列でなければならず、その各要素は左から右へと対応するターゲットに代入されます。
- 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。
- ターゲットが添字表記の場合: 参照されている一次語の式が値評価されます。まず、値は変更可能な配列オブジェクト (例えばリスト) か、マップオブジェクト (例えば辞書) でなければなりません。次に、添字表記の表す式が値評価されます。

一次語が変更可能な配列オブジェクト (例えばリスト) の場合、まず添字は整数でなければなりません。添字が負数の場合、配列の長さが加算されます。添字は最終的に、配列の長さよりも小さな非負の整数でなくてはなりません。次に、添字をインデックスに持つ要素に非代入オブジェクトを代入してよいか、配列に問い合わせます。範囲を超えたインデックスに対しては `IndexError` が送出されます (添字指定された配列に代入を行っても、リスト要素の新たな追加はできません)。

一次語がマップオブジェクト (例えば辞書) の場合、まず添字はマップのキー型と互換性のある型でなくてはなりません。次に、添字を被代入オブジェクトに関連付けるようなキー/データの対を生成するようマップオブジェクトに問い合わせます。この操作では、既存のキー/値の対を同じキーと別の値で置き換えてもよく、(同じ値を持つキーが存在しない場合) 新たなキー/値の対を挿入してもかまいません。

- ターゲットがスライスの場合: 参照されている一次語の式が値評価されます。まず、値は変更可能な配列オブジェクト (例えばリスト) でなければなりません。被代入オブジェクトは同じ型を持った配列オブジェクトでなければなりません。次に、スライスの下境界と上境界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ゼロと配列の長さです。上下境界は整数にならなければなりません。いずれかの境界が負数になった場合、配列の長さが加算されます。最終的に、境界はゼロから配列の長さまでの内包になるようにクリップされます。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配列オブジェクトに問い合わせます。オブジェクトで許されている限り、スライスの長さは被代入配列の長さとは異なっていてもよく、この場合にはターゲット配列の長さが変更されます。

(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。)

警告: 代入の定義では、左辺値と右辺値がオーバーラップするような代入 (例えば、`'a, b = b, a'` を行くと、二つの変数を入れ替えます) を定義しても '安全 (safe)' に代入できますが、代入対象となる変数群の間でオーバーラップがある場合は安全ではありません! 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
print x
```

### 6.3.1 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 ::= target augop expression_list
augop                      ::=  "+=" | "-=" | "*=" | "/=" | "%=" | "**="
                              | ">>=" | "<<=" | "&=" | "^=" | "|="
```

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。

`x += 1` のような累算代入式は、`x = x + 1` のように書き換えてほぼ同様の動作にできますが、厳密に等価にはなりません。累算代入の方では、`x` は一度しか評価されません。また、実際の処理として、可能ならばインプレース (*in-place*) 演算が実行されます。これは、代入時に新たなオブジェクトを生成してターゲットに代入するのではなく、以前のオブジェクトの内容を変更するということです。

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、代入前の初期値は `getattr()` で取り出され、演算結果は `setattr()` で代入されます。二つのメソッドが同じ変数を参照するという必然性はないので注意してください。例えば:

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

のように、`getattr()` がクラス変数を参照していても、`setattr()` はインスタンス変数への書き込みを行ってしまいます。

## 6.4 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、例えば:



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

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

のように、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。

## 6.5 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を記述するよりも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。

名前に対して削除を行うと、ローカルまたはグローバル名前空間でのその名前の束縛を解除します。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) であるばあい、`NameError` 例外が送出されます。

ネストしたブロック中で自由変数 になっているローカル名前空間上の名前に対する削除は不正な操作になります

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)。

## 6.6 print 文

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

`print` は、式を逐次的に評価し、得られたオブジェクトを標準出力に書き出します。オブジェクトが文字列でなければ、まず文字列変換規則を使って文字列に変換され、次いで (得られた文字列か、オリジナルの文字列が) 書き出されます。出力系の現在の書き出し位置が行頭にあると考えられる場合を除き、各オブジェクトの出力前にスペースが一つ出力されます。行頭にある場合とは、(1) 標準出力にまだ何も書き出されていない場合、(2) 標準出力に最後に書き出された文字が `'\n'` である、または (3) 標準出力に対する最後の書き出し操作が `print` 文によるものではない場合、です。(こうした理由から、場合によっては空文字を標準出力に書き出すと便利ことがあります。) 注意: 組み込みのファイルオブジェクトでない、ファイルオブジェクトに似た動作をするオブジェクトでは、組み込みのファイルオブジェクトが持つ上記の性質を適切にエミュレートしていないことがあるため、当てにしないほうがよいでしょう。

`print` 文がカンマで終了していない限り、末尾には文字 `'\n'` が書き出されます。この仕様は、文に予約語 `print` がある場合のみの動作です。

標準出力は、組み込みモジュール `sys` 内で `stdout` という名前のファイルオブジェクトとして定義されています。該当するオブジェクトが存在しないか、オブジェクトに `write()` メソッドがない場合、`RuntimeError` 例外が送出されます。 .

`print` には、上で説明した構文の第二形式で定義されている拡張形式 があります。この形式は、“山形 `print` 表記 (print chevron)” と呼ばれます。この形式では、`>>` の直後にくる最初の式の値評価結果は“ファイル類似 (file-like)” なオブジェクト、とりわけ上で述べたように `write()` メソッドを持つオブジェクトでなければなりません。この拡張形式では、ファイルオブジェクトを指定する式よりも後ろの式が、指定

されたファイルオブジェクトに出力されます。最初の式の値評価結果が `None` になった場合、`sys.stdout` が出力ファイルとして使われます。

## 6.7 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。

ジェネレータ関数の場合には、`return` 文の中に `expression_list` を入れることはできません。ジェネレータ関数の処理コンテキストでは、単体の `return` はジェネレータ処理を終了し `StopIteration` を送出させることを示します。

## 6.8 yield 文

```
yield_stmt ::= "yield" expression_list
```

`yield` 文は、ジェネレータ関数 (generator function) を定義するときだけ使われ、かつジェネレータ関数の本体の中でだけ用いられます。関数定義中で `yield` 文を使うだけで、関数定義は通常の関数でなくジェネレータ関数になります。

ジェネレータ関数が呼び出されると、ジェネレータイテレータ (generator iterator)、一般的にはジェネレータ (generator) を返します。ジェネレータ関数の本体は、ジェネレータの `next()` が例外を発行するまで繰り返し呼び出して実行します。

`yield` 文が実行されると、現在のジェネレータの状態は凍結 (freeze) され、`expression_list` の値が `next()` の呼び出し側に返されます。ここでの“凍結”は、ローカルな変数への束縛、命令ポインタ (instruction pointer)、および内部実行スタック (internal evaluation stack) を含む、全てのローカルな状態が保存されることを意味します: すなわち、必要な情報を保存しておき、次に `next()` が呼び出された際に、関数が `yield` 文をあたかももう一つの外部呼出しであるかのように処理できるようにします。

`yield` 文は、`try ... finally` 構造の `try` 節中で使うことはできません。ジェネレータが常に実行再開されるとは限らないので、`finally` ブロックが常に実行される保証がないという問題があるためです。

注意: Python 2.2 では、`generators` 機能が有効になっている場合にのみ `yield` 文を使えます。Python 2.3 では、常に有効になっています。`__future__ import` 文を使うと、この機能を有効にできます:

```
from __future__ import generators
```

参考資料:

PEP 0255, “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

## 6.9 raise 文

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

式を伴わない場合、`raise` は現在のスコープで最終的に有効になっている式を再送出します。そのような式が現在のスコープに全くない場合、エラーを示す例外が送出されます。

それ以外の場合、`raise` は式を値評価して、三つのオブジェクトを取得します。このとき、`None` を省略された式の値として使います。最初の二つのオブジェクトは、例外の型 (*type*) と例外の値 (*value*) を決定するために用いられます。

最初のオブジェクトがインスタンスである場合、例外の型はインスタンスのクラスになり、インスタンス自体が例外の値になります。このとき第二のオブジェクトは `None` でなければなりません。

最初のオブジェクトがクラスの場合、例外の型になります。第二のオブジェクトは、例外の値を決めるために使われます: 第二のオブジェクトがインスタンスならば、そのインスタンスが例外の値になります。第二のオブジェクトがタブルの場合、クラスのコンストラクタに対する引数リストとして使われます; `None` なら、空の引数リストとして扱われ、それ以外の型ならコンストラクタに対する単一の引数として扱われます。このようにしてコンストラクタを呼び出して生成したインスタンスが例外の値になります。

第三のオブジェクトが存在し、かつ `None` でなければ、オブジェクトはトレースバック オブジェクトでなければなりません (3.2 節参照)。また、例外が発生した場所は現在の処理位置に置き換えられます。第三のオブジェクトが存在し、オブジェクトがトレースバックオブジェクトでも `None` でもなければ、`TypeError` 例外が送出されます。`raise` の三連式型は、`except` 節から透過的に例外を再送出するのに便利ですが、再送出すべき例外が現在のスコープで発生した最も新しいアクティブな例外である場合には、式なしの `raise` を使うよう推奨します。

例外に関する追加情報は 4.2 節にあります。また、例外処理に関する情報は 7.4 節にあります。

## 6.10 break 文

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

`break` 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には現れません。

`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 `else` 節に飛びます。

`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。

## 6.11 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、`try` 文の中には現れません。<sup>1</sup>

`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。

---

<sup>1</sup>`except` 節や `else` 節中に置くことはできます。`try` 文に置けないという制限は、実装側の不精によるもので、そのうち改善されることでしょう。

## 6.12 import 文

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

import 文は、(1) モジュールを探し、必要なら初期化 (initialize) する; (import 文のあるスコープにおける) ローカルな名前空間で名前を定義する、の二つの段階を踏んで初期化されます。第一形式 (from のない形式) は、上記の段階をリスト中にある各識別子に対して繰り返し実行していきます。from のある形式では、(1) を一度だけ行い、次いで (2) を繰り返し実行します。

組み込みモジュールや拡張モジュールの“初期化”は、ここでは初期化関数の呼び出しを意味します。モジュールは初期化を行うためにかならず初期化関数を提供しなければなりません (リファレンス実装では、関数名はモジュール名の前に “init” をつけたものになっています); Python で書かれたモジュールの“初期化”は、モジュール本体の実行を意味します。

Python 処理系は、すでに初期化済みのモジュールや、初期化中のモジュールをモジュール名でインデックス化したテーブルを維持しています。このテーブルは `sys.modules` からアクセスできます。モジュール名がこのテーブル内にあるなら、段階 (1) は完了しています。そうでなければ、処理系はモジュール定義の検索を開始します。モジュールが見つかった場合、モジュールを読み込み (load) ます。モジュール検索や読み込みプロセスの詳細は、実装やプラットフォームに依存します。一般的には、ある名前のモジュールを検索する際、まず同名の“組み込み (built-in)”モジュールを探し、次に `sys.path` で列挙されている場所を探します。

組み込みモジュールが見つかった場合、組み込みの初期化コードが実行され、段階 (1) を完結します。合致するファイルが見つからなかった場合、`ImportError` が送出されます。ファイルが見つかった場合、ファイルを構文解析して実行可能なコードブロックにします。構文エラーが起きた場合、`SyntaxError` が送出されます。それ以外の場合、まず指定された名前をもつ空のモジュールを作成し、モジュールテーブルに挿入します。次に、このモジュールの実行コンテキスト下でコードブロックを実行します。実行中に例外が発生すると、段階 (1) を終了 (terminate) します。

段階 (1) が例外を送出することなく完了したなら、段階 (2) を開始します。

import 文の第一形式は、ローカルな名前空間に置かれたモジュール名をモジュールオブジェクトに束縛し、import すべき次の識別子があればその処理に移ります。モジュール名の後ろに `as` がある場合、`as` の後ろの名前はモジュールのローカルな名前として使われます。

from 形式は、モジュール名の束縛を行いません: from 形式では、段階 (1) で見つかったモジュール内から、識別子リストの各名前を順に検索し、見つかったオブジェクトを識別子の名前でローカルな名前空間において束縛します。import の第一形式と同じように、`"as localname"` で別名を与えることができます。指定された名前が見つからない場合、`ImportError` が送出されます。識別子のリストを星印 (`*`) で置き換えると、モジュールで公開されている名前 (public name) 全てを import 文のある場所のローカルな名前空間に束縛します。。。

モジュールで公開されている名前 (*public names*) は、モジュールの名前空間内にある `__all__` という名前の変数を調べて決定します; `__all__` が定義されている場合、`__all__` はモジュールで定義されていたり、import されているような名前の文字列からなる配列でなければなりません。`__all__` 内にある名前は、全て公開された名前であり、実在するものとみなされます。`__all__` が定義されていない場合、モジュールの名前空間に見つかった名前、アンダースコア文字 (`'_'`) で始まっていない全ての名前が公開された名前になります。`__all__` には、公開されている API 全てを入れなければなりません。`__all__` には、(モジュール内で import されて使われているライブラリモジュールのように) API を構成しない要素を

意に反して公開してしまうのを避けるという意図があります。

‘\*’を使った `from` 形式は、モジュールのスコープ内だけに作用します。関数内でワイルドカードの `import` 文 — `import *` — を使い、関数が自由変数を伴うネストされたブロックであったり、ブロックを含んでいる場合、コンパイラは `SyntaxError` を送出します。

階層的なモジュール名: モジュール名に一つまたはそれ以上のドットが入っている場合、モジュール検索パスは違った扱われ方をします。最後のドットまでの各識別子からなる配列は、“パッケージ (package)”を見つけるために使われます; 次に、パッケージ内から各識別子が検索されます。パッケージとは、一般には `sys.path` 上のディレクトリのサブディレクトリで、‘`__init__.py`’ ファイルを持つものです。[XXX この説明については、ここでは今のところこれ以上詳しく書けません; 詳細や、パッケージ内モジュールの検索がどのように行われるかは、<http://www.python.org/doc/essays/packages.html> を参照してください]

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 `__import__()` が提供されています; 詳細は、*Python* ライブラリリファレンス の組み込み関数 を参照してください。

### 6.12.1 future 文 (future statement)

*future* 文 は、将来の特定の 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、言語仕様に非互換性がもたらされるような、将来の Python のバージョンに容易に移行できるよう意図されています。future 文によって、新たな機能が標準化されたリリースが出される前に、その機能をモジュール単位で使えるようにします。

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

- the module docstring (if any),
- comments,
- blank lines, and
- other future statements.

です。

Python 2.3 が feature 文で新たに認識するようになった機能は、‘generators’、‘division’、および ‘nested\_scopes’ です。‘generators’ および ‘nested\_scopes’ は Python 2.3 では常に有効になっているので、冗長な機能名といえます。

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:

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

には、何ら特殊な意味はないので注意してください。

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。

future 文の入ったモジュール M 内で使われている `exec` 文、組み込み関数 `compile()` や `execfile()` によってコンパイルされるコードは、デフォルトの設定では、future 文に関係する新たな構文や意味付けを使うようになっています。Python 2.2 からは、この仕様を `compile()` のオプション引数で制御できるようになりました — 詳細はライブラリリファレンスの `compile()` に関するドキュメントを参照してください。

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。

## 6.13 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、for ループのループ制御ターゲットや、class 定義、関数定義、import 文内で仮引数として使ってはなりません。

(現在の実装では、後ろ二つの制限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、exec 文内に入っている global 文は、exec 文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、exec 文内に含まれているコードは、exec 文を含むコード内での global 文に影響を受けません。同様のことが、関数 `eval()`、`execfile()`、および `compile()` にも当てはまります。

## 6.14 exec 文

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



この文は、Python コードの動的な実行をサポートします。最初の式の値評価結果は文字列か、開かれたファイルオブジェクトか、コードオブジェクトでなければなりません。文字列の場合、一連の Python 実行文として解析し、(構文エラーが生じない限り) 実行します。開かれたファイルであれば、ファイルを EOF まで読んで解析し、実行します。コードオブジェクトなら、単にオブジェクトを実行します。

いずれの場合でも、オプションの部分が省略されると、コードは現在のスコープ内で実行されます。in の後ろに一つだけ式を指定する場合、その式は辞書でなくてはならず、グローバル変数とローカル変数の両方に使われます。二つの式が与えられた場合、双方ともに辞書でなくてはならず、それぞれグローバル変数とローカル変数として使われます。

exec の副作用として実行されるコードで設定された変数名に対応する名前の他に、追加のキーを辞書に追加することがあります。例えば、現在の実装では、組み込みモジュール `__builtin__` の辞書に対する参照を、`__builtins__` (!) というキーで追加することがあります。

プログラマのためのヒント: 式の動的な評価は、組み込み関数 `eval()` でサポートされています。組み込み関数 `globals()` および `locals()` は、それぞれ現在のグローバル辞書とローカル辞書を返すので、`exec` に渡して使うと便利です。



## 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` は例外処理かつ/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。関数とクラス定義もまた、構文法的には複合文です。

複合文は、一つまたはそれ以上の '節 (clause)' からなります。一つの節は、ヘッダと 'スイート (suite)' からなります。特定の複合文を構成する節のヘッダ部分は、全て同じインデントレベルになります。各々の節ヘッダ行は一意に識別されるキーワードから始まり、コロンの終わります。スイートは、ヘッダのコロンの後ろにセミコロンで区切られた一つまたはそれ以上の単純文を並べるか、ヘッダ行後の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、ネストされた複合文を入れることができます; 以下の文は、`else` 節がどの `if` 節に属するかがはっきりしないという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、セミコロンはコロンよりも強い結合を表すことにも注意してください。従って、以下の例では、`print` は全て実行されるか、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
                | funcdef
                | classdef
suite          ::= stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement      ::= stmt_list NEWLINE | compound_stmt
stmt_list      ::= simple_stmt (";" simple_stmt)* [";"]
```

文は常に `NEWLINE` か、その後に `DEDENT` が続いたもので終了することに注意してください。また、オプションの継続節は常にあるキーワードから始まり、このキーワードから複合文を開始することはできないため、曖昧さは存在しないことにも注意してください (Python では、'ぶら下がり (dangling else)' 問題を、ネストされた `if` 文はインデントさせること解決しています)。

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。

## 7.1 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" expression ":" suite
         ( "elif" 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、[5.10 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 (または、if 文の他の部分を実行したり、評価したりします) 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。

## 7.2 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" 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。

## 7.3 for 文

for 文は、配列 (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" expression_list ":" suite
           ["else" ":" suite]
```

式リストは一度だけ値評価されます; 結果はイテレーション可能なオブジェクトにならなければなりません。expression\_list の結果からイテレータが生成されます。次に、配列の各要素についてインデックスの小さい順に一度だけスイートを実行します。このとき配列内の要素が通常の代入規則を使ってターゲットリストに代入され、その後スイートが実行されます。全ての要素を使い切ると (配列が空の場合にはすぐに)、else 節があればそれが実行され、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。

スイートの中では、ターゲットリスト内の変数に代入を行えます; この代入によって、次に代入される要素に影響を及ぼすことはありません。

ループが終了してもターゲットリストは削除されませんが、配列が空の場合には、ループでの代入は全く行われません。ヒント: 組み込み関数 range() は、Pascal 言語における for i := a to b do の効果をエミュレートするのに適した数列を返します; すなわち、range(3) はリスト [0, 1, 2] を返します。

警告: ループ中の配列の変更には微妙な問題があります (これは変更可能な配列、すなわちリストで起こります)。どの要素が次に使われるかを追跡するために、内部的なカウンタが使われており、このカウンタ

は反復処理を行うごとに加算されます。このカウンタが配列の長さに達すると、ループは終了します。このことは、スイート中で配列から現在の (または以前の) 要素を除去すると、(次の要素のインデックスは、すでに取り扱った要素のインデックスになるために) 次の要素が飛ばされることを意味します。同様に、スイート中で配列中の現在の要素以前に要素を挿入すると、ループ中で現在の要素が再度扱われることになります。こうした仕様は、厄介なバグになります。配列全体に相当するスライスを使って一時的なコピーを作ると、これを避けることができます。

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

## 7.4 try 文

try 文は、ひとまとめの文に対して、例外処理かつ/またはクリーンアップコードを指定します:

```
try_stmt      ::= try_exc_stmt | try_fin_stmt
try_exc_stmt  ::= "try" ":" suite
                  ("except" [expression ["," target]] ":" suite)+
                  ["else" ":" suite]
try_fin_stmt  ::= "try" ":" suite "finally" ":" suite
```

try 文には二つの形式: try...except および try...finally があります。これら二つの形式を混合することはできません (互いにネストすることはできます)。

try...except 形式では、一つまたはそれ以上の例外ハンドラ (except 節) を指定します。try 節内で全く例外が起きなければ、どの例外ハンドラも実行されません。try スイート内で例外が発生すると、例外ハンドラの検索が開始されます。この検索では、except 節を逐次調べて、発生した例外に合致するまで続けます。式を伴わない except 節を使う場合、最後に書かなければなりません; この except 節は全ての例外に合致します。式を伴う except 節に対しては、式が値評価され、返されたオブジェクトが例外と“互換である (compatible)”場合にその節が合致します。ある例外に対してオブジェクトが互換であるのは、オブジェクトがその例外のアイデンティティを持つオブジェクトであるか、(クラスの例外の場合) 例外の基底クラスであるか、例外と互換性のある要素が入ったタプルである場合です。同じ値を持つオブジェクトであるだけでなく、アイデンティティが合致しなければならないので注意してください。

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。

except 節のヘッダにある式を値評価するとき例外が発生すると、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)。

合致する except 節が見つかり、その except 節にターゲットが指定されている場合、ターゲットに例外のパラメタが代入され、その後 except 節のスイートが実行されます。except 節は全て実行可能なブロックを持っていなければなりません。ブロックの末尾に到達すると、通常は try 文全体の直後に実行を継続します。(このことは、同じ例外に対してネストした二つの例外ハンドラが存在し、内側のハンドラ内の try 節で例外が発生した場合、外側のハンドラが例外を処理できないことを意味します。)

except 節のスイートが実行される前に、例外に関する詳細が sys モジュール内の三つの変数に代入されます: sys.exc\_type は、例外を示すオブジェクトを受け取ります; sys.exc\_value は例外のパラメタを受け取ります; sys.exc\_traceback は、プログラム上の例外が発生した位置を識別するトレースバックオブジェクト (3.2 節参照) を受け取ります。これらの詳細はまた、関数 sys.exc\_info() を介して入手することもできます。この関数はタプル (exc\_type, exc\_value, exc\_traceback) を返します。ただしこの関数に対応する変数の使用は、スレッドを使ったプログラムで安全に使えないため撤廃されてい

まず、Python 1.5 からは、例外を処理した関数から戻るときに、以前の値 (関数呼び出し前の値) に戻されます。

オプションの `else` 節は、実行の制御が `try` 節の末尾に到達した場合に実行されます。<sup>1</sup> `else` 節内で起きた例外は、`else` 節に先行する `except` 節で処理されることはありません。

`try...finally` 形式では、‘クリーンアップ’ハンドラを指定します。まず `try` 節が実行されます。例外が全く発生しなければ、`finally` 節が実行されます。例外が `try` 節内で発生した場合、例外は一時的に保存され、`finally` が実行された後、保存されていた例外が再送出されます。`finally` 節で別の例外が送出されたり、`return` や `break` 節が実行された場合、保存されていた例外は失われます。`finally` 節での `continue` 文の使用は不正となります (理由は現在の実装上の問題にあります — この制限は将来解消されるかもしれません)。`finally` 節の実行中は、例外情報を取得することはできません。

`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 ‘抜け出る途中に (on the way out)’ 実行されます。

例外に関するその他の情報は 4.2 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、6.9 節にあります。

## 7.5 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 (3.2 節参照):

```
funcdef          ::= "def" funcname "(" [parameter_list] ")" ":" suite
parameter_list   ::= (defparameter ",")*
                    (  "*" identifier [, "*" identifier]
                    |  "*" identifier
                    |  defparameter [","] )
defparameter     ::= parameter ["=" expression]
sublist          ::= parameter ("," parameter)* [","]
parameter        ::= identifier | "(" sublist ")"
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プ) 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。

一つ以上のトップレベルのパラメタに `parameter = expression` の形式がある場合、関数は “デフォルトのパラメタ値 (default parameter values)” を持つといいます。デフォルト値を伴うパラメタに対しては、関数呼び出しの際に対応するパラメタが省略されると、パラメタの値はデフォルト値で置き換えられます。あるパラメタがデフォルト値を持つ場合、それ以後のパラメタは全てデフォルト値を持たなければなりません — これは文法的には表現されていない構文上の制限です。

デフォルトパラメタ値は関数定義を実行する際に値評価されます。これは、デフォルトパラメタの式は関数を定義するときただ一度だけ評価され、同じ “計算済みの” 値が全ての呼び出しで使われることを意味します。デフォルトパラメタ値がリストや辞書のような変更可能なオブジェクトである場合、この使用を理解しておくことは特に重要です: 関数でこのオブジェクトを (例えばリストに要素を追加して) 変更すると、実際のデフォルト値が変更されてしまいます。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、デフォルト値に `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

<sup>1</sup> 現在、制御が “末尾に到達する” のは、例外が発生したり、`return`、`continue`、または `break` 文が実行される場合を除きます。



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、[5.3.4](#) 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対して、固定引数、キーワード引数、デフォルト引数のいずれかから値を代入します。“\*identifier”形式が存在する場合、余った固定引数を受け取るタプルに初期化されます。この変数のデフォルト値は空のタプルです。“\*\*identifier”形式が存在する場合、余ったキーワード引数を受け取るタプルに初期化されます。デフォルト値は空の辞書です。

式で直接使うために、無名関数(名前に束縛されていない関数)を作成することも可能です。無名関数の作成には、[5.11](#) 節で記述されているラムダ形式(lambda form)を使います。ラムダ形式は、単純化された関数定義を行うための略記法にすぎません; “def”文で定義された関数は、ラムダ形式で定義された関数と全く同様に引渡ししたり、他の名前に代入したりできます。実際には、“def”形式は複数の式を実行できるという点でより強力です。

プログラマのための注釈: 関数は一級の(first-class)オブジェクトです。関数定義内で“def”形式を実行すると、戻り値として返したり引き渡したりできるローカルな関数を定義します。ネストされた関数内で自由変数を使うと、def文の入っている関数のローカル変数にアクセスすることができます。詳細は[4.1](#) 節を参照してください。

## 7.6 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([3.2](#) 節参照):

```
classdef      ::= "class" classname [inheritance] ":" suite
inheritance   ::= "(" expression_list ")"
classname    ::= identifier
```

クラス定義は実行可能な文です。クラス定義では、まず継承リストがあればそれを評価します。継承リストの各要素の値評価結果はクラスオブジェクトでなければなりません。次にクラスのスイートが新たな実行フレーム内で、新たな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([4.1](#) 節を参照してください)。(通常、スイートには関数定義のみが含まれます) クラスのスイートを実行し終わると、実行フレームは無視されますが、ローカルな名前空間は保存されます。次に、基底クラスの継承リストを使ってクラスオブジェクトが生成され、ローカルな名前空間を属性値辞書として保存します。最後に、もとのローカルな名前空間において、クラス名がこのクラスオブジェクトに束縛されます。

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変数です; クラス変数は全てのインスタンス間で共有されます。インスタンス変数を定義するには、\_\_init\_\_() メソッドや他のメソッド中で変数に値を与えます。クラス変数もインスタンス変数も“self.name”表記でアクセスすることができます。この表記でアクセスする場合、インスタンス変数は同名のクラス変数を隠蔽します。変更不能な値をもつクラス変数は、インスタンス変数のデフォルト値として使えます。



# 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。

## 8.1 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 プログラムについての概念を持っておくに役に立ちます。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)、`__builtin__` (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。

UNIX の環境下では、完全なプログラムをインタプリタに渡すには三通りの方法があります: 第一は、`-c string` コマンドラインオプションを使う方法、第二はファイルを第一コマンドライン引数として指定する方法、そして最後は標準入力から入力する方法です。ファイルや標準入力 (tty (端末) デバイスの場合、インタプリタは対話モードに入ります; そうでない場合、ファイルを完全なプログラムとして実行します。

## 8.2 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;

## 8.3 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。

## 8.4 式入力

式入力には二つの形式があります。双方とも、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*
```

input() で読み込まれる入力行は、以下の形式をとらなければなりません:

```
input_input ::= expression_list NEWLINE
```

注意: 文としての解釈を行わない‘生の (raw)’ 入力行を読み出すためには、組み込み関数 raw\_input() や、ファイルオブジェクトの readline() メソッドを使うことができます。

# 歴史とライセンス

## A.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <http://www.cwi.nl/>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <http://www.cnri.reston.va.us/>) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <http://www.zope.com/>). In 2001, the Python Software Foundation (PSF, see <http://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see <http://www.opensource.org/> for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 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
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.3	2.3.2	2004	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes

注意: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

## A.2 Terms and conditions for accessing or otherwise using Python

### **PSF LICENSE AGREEMENT FOR PYTHON 2.3.4**

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.3.4 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.3.4 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2003 Python Software Foundation; All Rights Reserved" are retained in Python 2.3.4 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.3.4 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.3.4.
4. PSF is making Python 2.3.4 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.3.4 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3.4 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3.4, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.3.4, Licensee agrees to be bound by the terms and conditions of this License Agreement.

### **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.

#### **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 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.

ACCEPT

**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.

# 日本語訳について

## B.1 このドキュメントについて

この文書は、Python ドキュメント翻訳プロジェクトによる Python Language Reference の日本語訳版です。日本語訳に対する質問や提案などがありましたら、Python ドキュメント翻訳プロジェクトのメーリングリスト

<http://www.python.jp/mailman/listinfo/python-doc-jp>

または、プロジェクトのバグ管理ページ

[http://sourceforge.jp/tracker/?atid=116\&group\\_id=11\&func=browse](http://sourceforge.jp/tracker/?atid=116\&group_id=11\&func=browse)

までご報告ください。

## B.2 翻訳者一覧 (敬称略)

2.3.2 和訳 Yasushi Masuda <y.masuda at acm.org> (February 3, 2004)

2.3.3 差分 Yasushi Masuda (May 31, 2004)

2.3.4 差分 Yasushi Masuda (September 20, 2004)



# 索引

## Symbols

`__abs__()` (numeric object のメソッド), 34  
`__add__()` (numeric object のメソッド), 32  
`__add__()` (sequence object method), 30  
`__all__` (optional module attribute), 63  
`__and__()` (numeric object のメソッド), 32  
`__bases__` (class attribute), 20  
`__builtin__` (組み込みモジュール), 65, 73  
`__builtins__`, 65  
`__call__()` (object のメソッド), 29  
`__call__()` (object method), 47  
`__class__` (instance attribute), 20  
`__cmp__()` (object のメソッド), 25  
`__cmp__()` (object method), 25  
`__coerce__()` (numeric object のメソッド), 34  
`__coerce__()` (numeric object method), 30  
`__complex__()` (numeric object のメソッド), 34  
`__contains__()` (container object のメソッド), 31  
`__contains__()` (mapping object method), 30  
`__contains__()` (sequence object method), 30  
`__debug__`, 56  
`__del__()` (object のメソッド), 23  
`__delattr__()` (object のメソッド), 26  
`__delete__()` (object のメソッド), 27  
`__delitem__()` (container object のメソッド), 30  
`__delslice__()` (sequence object のメソッド), 31  
`__dict__` (class attribute), 20  
`__dict__` (function attribute), 17  
`__dict__` (instance attribute), 20, 26  
`__dict__` (module attribute), 19  
`__div__()` (numeric object のメソッド), 33  
`__divmod__()` (numeric object のメソッド), 32  
`__doc__` (class attribute), 20  
`__doc__` (function attribute), 17  
`__doc__` (method attribute), 17  
`__doc__` (module attribute), 19  
`__eq__()` (object のメソッド), 24  
`__file__` (module attribute), 19  
`__float__()` (numeric object のメソッド), 34  
`__floordiv__()` (numeric object のメソッド), 32  
`__ge__()` (object のメソッド), 24  
`__get__()` (object のメソッド), 26  
`__getattr__()` (object のメソッド), 25  
`__getattribute__()` (object のメソッド), 26  
`__getitem__()` (container object のメソッド), 30  
`__getitem__()` (mapping object method), 23  
`__getslice__()` (sequence object のメソッド), 31  
`__gt__()` (object のメソッド), 24  
`__hash__()` (object のメソッド), 25  
`__hex__()` (numeric object のメソッド), 34  
`__iadd__()` (numeric object のメソッド), 33  
`__iadd__()` (sequence object method), 30  
`__iand__()` (numeric object のメソッド), 33  
`__idiv__()` (numeric object のメソッド), 33  
`__ifloordiv__()` (numeric object のメソッド), 33  
`__ilshift__()` (numeric object のメソッド), 33  
`__imod__()` (numeric object のメソッド), 33  
`__import__()` (組み込み関数), 63  
`__imul__()` (numeric object のメソッド), 33  
`__imul__()` (sequence object method), 30  
`__init__()` (object のメソッド), 23  
`__init__()` (object method), 19  
`__init__.py`, 63  
`__int__()` (numeric object のメソッド), 34  
`__invert__()` (numeric object のメソッド), 34

`__ior__()` (numeric object のメソッド), 33  
`__ipow__()` (numeric object のメソッド), 33  
`__irshift__()` (numeric object のメソッド), 33  
`__isub__()` (numeric object のメソッド), 33  
`__iter__()` (container object のメソッド), 30  
`__iter__()` (sequence object method), 30  
`__itruediv__()` (numeric object のメソッド), 33  
`__ixor__()` (numeric object のメソッド), 33  
`__le__()` (object のメソッド), 24  
`__len__()` (container object のメソッド), 30  
`__len__()` (mapping object method), 25  
`__long__()` (numeric object のメソッド), 34  
`__lshift__()` (numeric object のメソッド), 32  
`__lt__()` (object のメソッド), 24  
`__main__` (組み込みモジュール), 38, 73  
`__metaclass__` (のデータ), 29  
`__mod__()` (numeric object のメソッド), 32  
`__module__` (class attribute), 20  
`__module__` (function attribute), 17  
`__module__` (method attribute), 17  
`__mul__()` (numeric object のメソッド), 32  
`__mul__()` (sequence object method), 30  
`__name__` (class attribute), 20  
`__name__` (function attribute), 17  
`__name__` (method attribute), 17  
`__name__` (module attribute), 19  
`__ne__()` (object のメソッド), 24  
`__neg__()` (numeric object のメソッド), 34  
`__nonzero__()` (object のメソッド), 25  
`__nonzero__()` (object method), 30  
`__oct__()` (numeric object のメソッド), 34  
`__or__()` (numeric object のメソッド), 32  
`__pos__()` (numeric object のメソッド), 34  
`__pow__()` (numeric object のメソッド), 32  
`__radd__()` (numeric object のメソッド), 33  
`__radd__()` (sequence object method), 30  
`__rand__()` (numeric object のメソッド), 33  
`__rcmp__()` (object のメソッド), 25  
`__rdiv__()` (numeric object のメソッド), 33  
`__rdivmod__()` (numeric object のメソッド), 33  
`__repr__()` (object のメソッド), 24  
`__rfloordiv__()` (numeric object のメソッド), 33  
`__rlshift__()` (numeric object のメソッド), 33  
`__rmod__()` (numeric object のメソッド), 33  
`__rmul__()` (numeric object のメソッド), 33

`__rmul__()` (sequence object method), 30  
`__ror__()` (numeric object のメソッド), 33  
`__rpow__()` (numeric object のメソッド), 33  
`__rrshift__()` (numeric object のメソッド), 33  
`__rshift__()` (numeric object のメソッド), 32  
`__rsub__()` (numeric object のメソッド), 33  
`__rtruediv__()` (numeric object のメソッド), 33  
`__rxor__()` (numeric object のメソッド), 33  
`__set__()` (object のメソッド), 27  
`__setattr__()` (object のメソッド), 26  
`__setattr__()` (object method), 26  
`__setitem__()` (container object のメソッド), 30  
`__setslice__()` (sequence object のメソッド), 31  
`__slots__` (のデータ), 28  
`__str__()` (object のメソッド), 24  
`__sub__()` (numeric object のメソッド), 32  
`__truediv__()` (numeric object のメソッド), 33  
`__unicode__()` (object のメソッド), 25  
`__xor__()` (numeric object のメソッド), 32

## A

`abs()` (組み込み関数), 34  
addition, 49  
and  
    bit-wise, 49  
and  
    演算子, 51  
    operator, 51  
anonymous  
    function, 52  
`append()` (sequence object method), 30  
argument  
    function, 17  
arithmetic  
    conversion, 41  
    operation, binary, 48  
    operation, unary, 48  
array (標準モジュール), 16  
ASCII, 2, 8, 9, 12, 16  
`assert`  
    実行文, 55  
    statement, 55  
`AssertionError`  
    exception, 56



- 例外, 56
- assignment
  - attribute, 56, 57
  - augmented, 58
  - class attribute, 20
  - class instance attribute, 20
  - slicing, 57
  - statement, 16, 56
  - subscription, 57
  - target list, 56
- atom, 41
- attribute, 14
  - assignment, 56, 57
  - assignment, class, 20
  - assignment, class instance, 20
  - class, 20
  - class instance, 20
  - deletion, 59
  - generic special, 14
  - reference, 44
  - special, 14
- AttributeError
  - exception, 44
  - 例外, 44
- augmented
  - assignment, 58

## B

- back-quotes, 24, 43
- backslash character, 4
- backward
  - quotes, 24, 43
- binary
  - arithmetic operation, 48
  - bit-wise operation, 49
- binding
  - global name, 64
  - name, 37, 56, 62, 70, 71
- bit-wise
  - and, 49
  - operation, binary, 49
  - operation, unary, 48
  - or, 49
  - xor, 49
- blank line, 5
- block, 37
  - code, 37

- BNF, 1
- Boolean
  - object, 15
  - オブジェクト, 15
  - operation, 51
- break
  - 実行文, 61, 68, 70
  - statement, 61, 68, 70
- bsddb (標準モジュール), 17
- built-in
  - method, 19
  - module, 62
- built-in function
  - call, 47
  - object, 19, 47
  - オブジェクト, 19, 47
- built-in method
  - call, 47
  - object, 19, 47
  - オブジェクト, 19, 47
- byte, 16
- bytecode, 21

## C

- C, 8
  - language, 14, 15, 19, 49
- call, 45
  - built-in function, 47
  - built-in method, 47
  - class instance, 47
  - class object, 19, 20, 47
  - function, 17, 47
  - instance, 29, 47
  - method, 47
  - procedure, 55
  - user-defined function, 47
- callable
  - object, 17, 45
  - オブジェクト, 17, 45
- chaining
  - comparisons, 50
- character, 16, 44
- character set, 16
- chr ( ) (組み込み関数), 16
- class
  - attribute, 20
  - attribute assignment, 20

- constructor, [23](#)
- definition, [60](#), [71](#)
- instance, [20](#)
- name, [71](#)
- object, [19](#), [20](#), [47](#), [71](#)
- オブジェクト, [19](#), [20](#), [47](#), [71](#)
- class
  - 実行文, [71](#)
  - statement, [71](#)
- class instance
  - attribute, [20](#)
  - attribute assignment, [20](#)
  - call, [47](#)
  - object, [19](#), [20](#), [47](#)
  - オブジェクト, [19](#), [20](#), [47](#)
- class object
  - call, [19](#), [20](#), [47](#)
- clause, [67](#)
- clear() (mapping object method), [30](#)
- cmp() (組み込み関数), [25](#)
- co\_argcount (code object attribute), [21](#)
- co\_cellvars (code object attribute), [21](#)
- co\_code (code object attribute), [21](#)
- co\_consts (code object attribute), [21](#)
- co\_filename (code object attribute), [21](#)
- co\_firstlineno (code object attribute), [21](#)
- co\_flags (code object attribute), [21](#)
- co\_freevars (code object attribute), [21](#)
- co\_lnotab (code object attribute), [21](#)
- co\_name (code object attribute), [21](#)
- co\_names (code object attribute), [21](#)
- co\_nlocals (code object attribute), [21](#)
- co\_stacksize (code object attribute), [21](#)
- co\_varnames (code object attribute), [21](#)
- code
  - block, [37](#)
  - object, [21](#)
  - オブジェクト, [21](#)
- code block, [62](#)
- comma, [42](#)
  - trailing, [52](#), [59](#)
- command line, [73](#)
- comment, [4](#)
- comparison, [49](#)
  - string, [16](#)
- comparisons, [25](#)
  - chaining, [50](#)

- compile() (組み込み関数), [64](#)
- complex
  - literal, [10](#)
  - number, [15](#)
  - object, [15](#)
  - オブジェクト, [15](#)
- complex() (組み込み関数), [34](#)
- compound
  - statement, [67](#)
- comprehensions
  - list, [42](#), [43](#)
- constant, [7](#)
- constructor
  - class, [23](#)
- container, [14](#), [20](#)
- continue
  - 実行文, [61](#), [68](#), [70](#)
  - statement, [61](#), [68](#), [70](#)
- conversion
  - arithmetic, [41](#)
  - string, [24](#), [43](#), [55](#)
- copy() (mapping object method), [30](#)
- count() (sequence object method), [30](#)

## D

- dangling
  - else, [67](#)
- data, [13](#)
  - type, [14](#)
  - type, immutable, [42](#)
- datum, [43](#)
- dbm (標準モジュール), [17](#)
- decimal literal, [10](#)
- DEDENT token, [5](#), [67](#)
- def
  - 実行文, [70](#)
  - statement, [70](#)
- default
  - parameter value, [70](#)
- definition
  - class, [60](#), [71](#)
  - function, [60](#), [70](#)
- del
  - 実行文, [16](#), [23](#), [59](#)
  - statement, [16](#), [23](#), [59](#)
- delete, [16](#)
- deletion

- attribute, [59](#)
- target, [59](#)
- target list, [59](#)

delimiters, [12](#)

destructor, [23](#), [57](#)

dictionary

- display, [43](#)
- object, [17](#), [20](#), [25](#), [43](#), [44](#), [57](#)
- オブジェクト, [17](#), [20](#), [25](#), [43](#), [44](#), [57](#)

display

- dictionary, [43](#)
- list, [42](#)
- tuple, [42](#)

division, [48](#)

divmod() (組み込み関数), [32](#), [33](#)

documentation string, [21](#)

## E

EBCDIC, [16](#)

elif

- keyword, [68](#)
- 予約語, [68](#)

Ellipsis

- object, [14](#)
- オブジェクト, [14](#)

else

- dangling, [67](#)

else

- keyword, [61](#), [68](#), [70](#)
- 予約語, [61](#), [68](#), [70](#)

empty

- list, [43](#)
- tuple, [16](#), [42](#)

environment, [37](#)

演算子

- and, [51](#)
- in, [51](#)
- is, [51](#)
- is not, [51](#)
- not, [51](#)
- not in, [51](#)
- or, [51](#)

error handling, [39](#)

errors, [39](#)

escape sequence, [8](#)

eval() (組み込み関数), [64](#), [65](#), [74](#)

evaluation

order, [52](#)

exc\_info (in module sys), [22](#)

exc\_traceback (in module sys), [22](#), [70](#)

exc\_type (in module sys), [70](#)

exc\_value (in module sys), [70](#)

except

- keyword, [69](#)

- 予約語, [69](#)

exception, [39](#), [61](#)

- AssertionError, [56](#)

- AttributeError, [44](#)

- handler, [22](#)

- ImportError, [62](#)

- NameError, [42](#)

- raising, [61](#)

- RuntimeError, [59](#)

- StopIteration, [60](#)

- SyntaxError, [62](#)

- TypeError, [48](#)

- ValueError, [49](#)

- ZeroDivisionError, [48](#)

exception handler, [39](#)

exclusive

- or, [49](#)

exec

- 実行文, [64](#)

- statement, [64](#)

execfile() (組み込み関数), [64](#)

execution

- frame, [37](#), [71](#)

- restricted, [38](#)

- stack, [22](#)

execution model, [37](#)

expression, [41](#)

- lambda, [52](#)

- list, [52](#), [55](#), [56](#)

- statement, [55](#)

extend() (sequence object method), [30](#)

extended

- slicing, [45](#)

extended print statement, [59](#)

extended slicing, [16](#)

extension

- filename, [62](#)

- module, [14](#)

## F

- `f_back` (frame attribute), [21](#)
- `f_builtins` (frame attribute), [21](#)
- `f_code` (frame attribute), [21](#)
- `f_exc_traceback` (frame attribute), [22](#)
- `f_exc_type` (frame attribute), [22](#)
- `f_exc_value` (frame attribute), [22](#)
- `f_globals` (frame attribute), [21](#)
- `f_lasti` (frame attribute), [21](#)
- `f_lineno` (frame attribute), [22](#)
- `f_locals` (frame attribute), [21](#)
- `f_restricted` (frame attribute), [21](#)
- `f_trace` (frame attribute), [22](#)
- `False`, [15](#)
- `file`
  - object, [20](#), [74](#)
  - オブジェクト, [20](#), [74](#)
- `filename`
  - extension, [62](#)
- `finally`
  - keyword, [60](#), [61](#), [70](#)
  - 予約語, [60](#), [61](#), [70](#)
- `float()` (組み込み関数), [34](#)
- floating point
  - number, [15](#)
  - object, [15](#)
  - オブジェクト, [15](#)
- floating point literal, [10](#)
- `for`
  - 実行文, [61](#), [68](#)
  - statement, [61](#), [68](#)
- `form`
  - `lambda`, [52](#), [71](#)
- `frame`
  - execution, [37](#), [71](#)
  - object, [21](#)
  - オブジェクト, [21](#)
- `free`
  - variable, [37](#), [59](#)
- `from`
  - 実行文, [37](#), [63](#)
  - keyword, [62](#), [63](#)
  - statement, [37](#), [63](#)
  - 予約語, [62](#), [63](#)
- `func_closure` (function attribute), [17](#)
- `func_code` (function attribute), [17](#)

- `func_defaults` (function attribute), [17](#)
- `func_dict` (function attribute), [17](#)
- `func_doc` (function attribute), [17](#)
- `func_globals` (function attribute), [17](#)
- `function`
  - anonymous, [52](#)
  - argument, [17](#)
  - call, [17](#), [47](#)
  - call, user-defined, [47](#)
  - definition, [60](#), [70](#)
  - generator, [60](#)
  - name, [70](#)
  - object, [17](#), [19](#), [47](#), [70](#)
  - オブジェクト, [17](#), [19](#), [47](#), [70](#)
  - user-defined, [17](#)
- `future`
  - statement, [63](#)

## G

- garbage collection, [13](#)
- `gdbm` (標準モジュール), [17](#)
- `generator`
  - function, [18](#), [60](#)
  - iterator, [18](#), [60](#)
  - object, [21](#)
  - オブジェクト, [21](#)
- `generic`
  - special attribute, [14](#)
- `get()` (mapping object method), [30](#)
- `global`
  - name binding, [64](#)
  - namespace, [17](#)
- `global`
  - 実行文, [57](#), [59](#), [64](#)
  - statement, [57](#), [59](#), [64](#)
- `globals()` (組み込み関数), [65](#)
- grammar, [1](#)
- grouping, [5](#)

## H

- handle an exception, [39](#)
- `handler`
  - exception, [22](#)
- `has_key()` (mapping object method), [30](#)
- hash character, [4](#)
- `hex()` (組み込み関数), [34](#)
- hexadecimal literal, [10](#)

- hierarchical
  - module names, 63
- hierarchy
  - type, 14
- I
- id() (組み込み関数), 13
- identifier, 6, 41
- identity
  - test, 51
- identity of an object, 13
- if
  - 実行文, 68
  - statement, 68
- im\_class (method attribute), 18
- im\_func (method attribute), 17, 18
- im\_self (method attribute), 17, 18
- imaginary literal, 10
- immutable
  - data type, 42
  - object, 16, 42, 43
  - オブジェクト, 16
- immutable object, 13
- immutable sequence
  - object, 16
  - オブジェクト, 16
- import
  - 実行文, 19, 62
  - statement, 19, 62
- ImportError
  - exception, 62
  - 例外, 62
- in
  - 演算子, 51
  - keyword, 68
  - operator, 51
  - 予約語, 68
- inclusive
  - or, 49
- INDENT token, 5
- indentation, 5
- index operation, 15
- index() (sequence object method), 30
- indices() (slice のメソッド), 22
- inheritance, 71
- initialization
  - module, 62
- input, 74
  - raw, 74
- input() (組み込み関数), 74
- insert() (sequence object method), 30
- instance
  - call, 29, 47
  - class, 20
  - object, 19, 20, 47
  - オブジェクト, 19, 20, 47
- int() (組み込み関数), 34
- integer, 16
  - object, 15
  - オブジェクト, 15
  - representation, 15
- integer literal, 10
- interactive mode, 73
- internal type, 20
- interpreter, 73
- inversion, 48
- invocation, 17
- is
  - 演算子, 51
  - operator, 51
- is not
  - 演算子, 51
  - operator, 51
- item
  - sequence, 44
  - string, 44
- item selection, 15
- items() (mapping object method), 30
- iteritems() (mapping object method), 30
- iterkeys() (mapping object method), 30
- intervalues() (mapping object method), 30
- J
- Java
  - language, 15
- 実行文
  - assert, 55
  - break, 61, 68, 70
  - class, 71
  - continue, 61, 68, 70
  - def, 70
  - del, 16, 23, 59
  - exec, 64
  - for, 61, 68

- from, [37](#), [63](#)
- global, [57](#), [59](#), [64](#)
- if, [68](#)
- import, [19](#), [62](#)
- pass, [58](#)
- print, [24](#), [59](#)
- raise, [61](#)
- return, [60](#), [70](#)
- try, [22](#), [69](#)
- while, [61](#), [68](#)
- yield, [60](#)

## K

- key, [43](#)
- key/datum pair, [43](#)
- keys() (mapping object method), [30](#)
- keyword, [7](#)
  - elif, [68](#)
  - else, [61](#), [68](#), [70](#)
  - except, [69](#)
  - finally, [60](#), [61](#), [70](#)
  - from, [62](#), [63](#)
  - in, [68](#)

## L

- lambda
  - expression, [52](#)
  - form, [52](#), [71](#)
- language
  - C, [14](#), [15](#), [19](#), [49](#)
  - Java, [15](#)
  - Pascal, [68](#)
- last\_traceback (in module sys), [22](#)
- leading whitespace, [5](#)
- len() (組み込み関数), [15](#), [17](#), [30](#)
- lexical analysis, [3](#)
- lexical definitions, [2](#)
- line continuation, [4](#)
- line joining, [3](#), [4](#)
- line structure, [3](#)
- list
  - assignment, target, [56](#)
  - comprehensions, [42](#), [43](#)
  - deletion target, [59](#)
  - display, [42](#)
  - empty, [43](#)
  - expression, [52](#), [55](#), [56](#)

- object, [16](#), [43](#), [44](#), [57](#)
- オブジェクト, [16](#), [43](#), [44](#), [57](#)
- target, [56](#), [68](#)

- literal, [7](#), [42](#)
- locals() (組み込み関数), [65](#)
- logical line, [3](#)
- long() (組み込み関数), [34](#)
- long integer
  - object, [15](#)
  - オブジェクト, [15](#)
- long integer literal, [10](#)
- loop
  - over mutable sequence, [69](#)
  - statement, [61](#), [68](#)
- loop control
  - target, [61](#)

## M

- makefile() (socket method), [20](#)
- mangling
  - name, [42](#)
- mapping
  - object, [17](#), [20](#), [44](#), [57](#)
  - オブジェクト, [17](#), [20](#), [44](#), [57](#)
- membership
  - test, [51](#)
- method
  - built-in, [19](#)
  - call, [47](#)
  - object, [17](#), [19](#), [47](#)
  - オブジェクト, [17](#), [19](#), [47](#)
  - user-defined, [17](#)
- minus, [48](#)
- module
  - built-in, [62](#)
  - extension, [14](#)
  - importing, [62](#)
  - initialization, [62](#)
  - name, [62](#)
  - names, hierarchical, [63](#)
  - namespace, [19](#)
  - object, [19](#), [44](#)
  - オブジェクト, [19](#), [44](#)
  - search path, [62](#)
  - user-defined, [62](#)
- modules (in module sys), [62](#)
- modulo, [48](#)



multiplication, [48](#)

mutable

object, [16](#), [56](#), [57](#)

オブジェクト, [16](#), [56](#), [57](#)

mutable object, [13](#)

mutable sequence

loop over, [69](#)

object, [16](#)

オブジェクト, [16](#)

## N

name, [6](#), [37](#), [41](#)

binding, [37](#), [56](#), [62](#), [70](#), [71](#)

binding, global, [64](#)

class, [71](#)

function, [70](#)

mangling, [42](#)

module, [62](#)

rebinding, [56](#)

unbinding, [59](#)

`NameError`

exception, [42](#)

例外, [42](#)

`NameError` (built-in exception), [37](#)

names

hierarchical module, [63](#)

private, [42](#)

namespace, [37](#)

global, [17](#)

module, [19](#)

negation, [48](#)

newline

suppression, [59](#)

`NEWLINE` token, [3](#), [67](#)

`None`

object, [14](#), [55](#)

オブジェクト, [14](#), [55](#)

`not`

演算子, [51](#)

operator, [51](#)

`not in`

演算子, [51](#)

operator, [51](#)

notation, [1](#)

`NotImplemented`

object, [14](#)

オブジェクト, [14](#)

`null`

operation, [58](#)

number, [10](#)

complex, [15](#)

floating point, [15](#)

numeric

object, [15](#), [20](#)

オブジェクト, [15](#), [20](#)

numeric literal, [10](#)

## O

object, [13](#)

Boolean, [15](#)

built-in function, [19](#), [47](#)

built-in method, [19](#), [47](#)

callable, [17](#), [45](#)

class, [19](#), [20](#), [47](#), [71](#)

class instance, [19](#), [20](#), [47](#)

code, [21](#)

complex, [15](#)

dictionary, [17](#), [20](#), [25](#), [43](#), [44](#), [57](#)

Ellipsis, [14](#)

file, [20](#), [74](#)

floating point, [15](#)

frame, [21](#)

function, [17](#), [19](#), [47](#), [70](#)

generator, [21](#)

immutable, [16](#), [42](#), [43](#)

immutable sequence, [16](#)

instance, [19](#), [20](#), [47](#)

integer, [15](#)

list, [16](#), [43](#), [44](#), [57](#)

long integer, [15](#)

mapping, [17](#), [20](#), [44](#), [57](#)

method, [17](#), [19](#), [47](#)

module, [19](#), [44](#)

mutable, [16](#), [56](#), [57](#)

mutable sequence, [16](#)

`None`, [14](#), [55](#)

`NotImplemented`, [14](#)

numeric, [15](#), [20](#)

plain integer, [15](#)

recursive, [43](#)

sequence, [15](#), [20](#), [44](#), [51](#), [57](#), [68](#)

slice, [30](#)

string, [16](#), [44](#)

traceback, [22](#), [61](#), [69](#)

- tuple, [16](#), [44](#), [52](#)
- unicode, [16](#)
- user-defined function, [17](#), [47](#), [70](#)
- user-defined method, [17](#)

オブジェクト

- Boolean, [15](#)
- built-in function, [19](#), [47](#)
- built-in method, [19](#), [47](#)
- callable, [17](#), [45](#)
- class, [19](#), [20](#), [47](#), [71](#)
- class instance, [19](#), [20](#), [47](#)
- code, [21](#)
- complex, [15](#)
- dictionary, [17](#), [20](#), [25](#), [43](#), [44](#), [57](#)
- Ellipsis, [14](#)
- file, [20](#), [74](#)
- floating point, [15](#)
- frame, [21](#)
- function, [17](#), [19](#), [47](#), [70](#)
- generator, [21](#)
- immutable, [16](#)
- immutable sequence, [16](#)
- instance, [19](#), [20](#), [47](#)
- integer, [15](#)
- list, [16](#), [43](#), [44](#), [57](#)
- long integer, [15](#)
- mapping, [17](#), [20](#), [44](#), [57](#)
- method, [17](#), [19](#), [47](#)
- module, [19](#), [44](#)
- mutable, [16](#), [56](#), [57](#)
- mutable sequence, [16](#)
- None, [14](#), [55](#)
- NotImplemented, [14](#)
- numeric, [15](#), [20](#)
- plain integer, [15](#)
- recursive, [43](#)
- sequence, [15](#), [20](#), [44](#), [51](#), [57](#), [68](#)
- slice, [30](#)
- string, [16](#), [44](#)
- traceback, [22](#), [61](#), [69](#)
- tuple, [16](#), [44](#), [52](#)
- unicode, [16](#)
- user-defined function, [17](#), [47](#), [70](#)
- user-defined method, [17](#)

oct( ) (組み込み関数), [34](#)

octal literal, [10](#)

open( ) (組み込み関数), [20](#)

operation

- binary arithmetic, [48](#)
- binary bit-wise, [49](#)
- Boolean, [51](#)
- null, [58](#)
- shifting, [49](#)
- unary arithmetic, [48](#)
- unary bit-wise, [48](#)

operator

- and, [51](#)
- in, [51](#)
- is, [51](#)
- is not, [51](#)
- not, [51](#)
- not in, [51](#)
- or, [51](#)
- overloading, [23](#)
- precedence, [52](#)

operators, [11](#)

or

- bit-wise, [49](#)
- exclusive, [49](#)
- inclusive, [49](#)

or

- 演算子, [51](#)
- operator, [51](#)

ord( ) (組み込み関数), [16](#)

order

- evaluation, [52](#)

output, [55](#), [59](#)

- standard, [55](#), [59](#)

OverflowError (built-in exception), [15](#)

overloading

- operator, [23](#)

## P

packages, [63](#)

parameter

- value, default, [70](#)

parenthesized form, [42](#)

parser, [3](#)

Pascal

- language, [68](#)

pass

- 実行文, [58](#)
- statement, [58](#)

path

- module search, 62
- physical line, 3, 4, 8
- plain integer
  - object, 15
  - オブジェクト, 15
- plain integer literal, 10
- plus, 48
- pop()
  - mapping object method, 30
  - sequence object method, 30
- popen() (in module os), 20
- popitem() (mapping object method), 30
- pow() (組み込み関数), 32, 33
- precedence
  - operator, 52
- primary, 44
- print
  - 実行文, 24, 59
  - statement, 24, 59
- private
  - names, 42
- procedure
  - call, 55
- program, 73
- Python Enhancement Proposals
  - PEP 0255, 60

## Q

- quotes
  - backward, 24, 43
  - reverse, 24, 43

## R

- raise
  - 実行文, 61
  - statement, 61
- raise an exception, 39
- raising
  - exception, 61
- range() (組み込み関数), 68
- raw input, 74
- raw\_input() (組み込み関数), 74
- readline() (file method), 74
- rebinding
  - name, 56
- recursive
  - object, 43

## オブジェクト, 43

- reference
  - attribute, 44
- reference counting, 13
- 例外
  - AssertionError, 56
  - AttributeError, 44
  - ImportError, 62
  - NameError, 42
  - RuntimeError, 59
  - StopIteration, 60
  - SyntaxError, 62
  - TypeError, 48
  - ValueError, 49
  - ZeroDivisionError, 48
- remove() (sequence object method), 30
- repr() (組み込み関数), 24, 44, 55
- representation
  - integer, 15
- reserved word, 7
- restricted
  - execution, 38
- return
  - 実行文, 60, 70
  - statement, 60, 70
- reverse
  - quotes, 24, 43
- reverse() (sequence object method), 30
- RuntimeError
  - exception, 59
  - 例外, 59

## S

- scope, 37
- search
  - path, module, 62
- sequence
  - item, 44
  - object, 15, 20, 44, 51, 57, 68
  - オブジェクト, 15, 20, 44, 51, 57, 68
- setdefault() (mapping object method), 30
- shifting
  - operation, 49
- simple
  - statement, 55
- singleton
  - tuple, 16

- slice, [44](#)
  - object, [30](#)
  - オブジェクト, [30](#)
- slice() (組み込み関数), [22](#)
- slicing, [15](#), [16](#), [44](#)
  - assignment, [57](#)
  - extended, [45](#)
- sort() (sequence object method), [30](#)
- space, [5](#)
- special
  - attribute, [14](#)
  - attribute, generic, [14](#)
- stack
  - execution, [22](#)
  - trace, [22](#)
- standard
  - output, [55](#), [59](#)
- Standard C, [8](#)
- standard input, [73](#)
- start (slice object attribute), [22](#), [45](#)
- statement
  - assert, [55](#)
  - assignment, [16](#), [56](#)
  - assignment, augmented, [58](#)
  - break, [61](#), [68](#), [70](#)
  - class, [71](#)
  - compound, [67](#)
  - continue, [61](#), [68](#), [70](#)
  - def, [70](#)
  - del, [16](#), [23](#), [59](#)
  - exec, [64](#)
  - expression, [55](#)
  - for, [61](#), [68](#)
  - from, [37](#), [63](#)
  - future, [63](#)
  - global, [57](#), [59](#), [64](#)
  - if, [68](#)
  - import, [19](#), [62](#)
  - loop, [61](#), [68](#)
  - pass, [58](#)
  - print, [24](#), [59](#)
  - raise, [61](#)
  - return, [60](#), [70](#)
  - simple, [55](#)
  - try, [22](#), [69](#)
  - while, [61](#), [68](#)
  - yield, [60](#)
- statement grouping, [5](#)
- stderr (in module sys), [20](#)
- stdin (in module sys), [20](#)
- stdio, [20](#)
- stdout (in module sys), [20](#), [59](#)
- step (slice object attribute), [22](#), [45](#)
- stop (slice object attribute), [22](#), [45](#)
- StopIteration
  - exception, [60](#)
  - 例外, [60](#)
- str() (組み込み関数), [24](#), [44](#)
- string
  - comparison, [16](#)
  - conversion, [24](#), [43](#), [55](#)
  - item, [44](#)
  - object, [16](#), [44](#)
  - オブジェクト, [16](#), [44](#)
  - Unicode, [8](#)
- string literal, [8](#)
- subscription, [15–17](#), [44](#)
  - assignment, [57](#)
- subtraction, [49](#)
- suite, [67](#)
- suppression
  - newline, [59](#)
- syntax, [1](#), [41](#)
- SyntaxError
  - exception, [62](#)
  - 例外, [62](#)
- sys (組み込みモジュール), [59](#), [62](#), [69](#), [73](#)
- sys.exc\_info, [22](#)
- sys.exc\_traceback, [22](#)
- sys.last\_traceback, [22](#)
- sys.modules, [62](#)
- sys.stderr, [20](#)
- sys.stdin, [20](#)
- sys.stdout, [20](#)
- SystemExit (built-in exception), [39](#)

## T

- tab, [5](#)
- target, [56](#)
  - deletion, [59](#)
  - list, [56](#), [68](#)
  - list assignment, [56](#)
  - list, deletion, [59](#)
  - loop control, [61](#)

tb\_frame (traceback attribute), [22](#)  
tb\_lasti (traceback attribute), [22](#)  
tb\_lineno (traceback attribute), [22](#)  
tb\_next (traceback attribute), [22](#)  
termination model, [39](#)  
test  
    identity, [51](#)  
    membership, [51](#)  
token, [3](#)  
trace  
    stack, [22](#)  
traceback  
    object, [22](#), [61](#), [69](#)  
    オブジェクト, [22](#), [61](#), [69](#)  
trailing  
    comma, [52](#), [59](#)  
triple-quoted string, [8](#)  
True, [15](#)  
try  
    実行文, [22](#), [69](#)  
    statement, [22](#), [69](#)  
tuple  
    display, [42](#)  
    empty, [16](#), [42](#)  
    object, [16](#), [44](#), [52](#)  
    オブジェクト, [16](#), [44](#), [52](#)  
    singleton, [16](#)  
type, [14](#)  
    data, [14](#)  
    hierarchy, [14](#)  
    immutable data, [42](#)  
type() (組み込み関数), [13](#)  
type of an object, [13](#)  
TypeError  
    exception, [48](#)  
    例外, [48](#)  
types, internal, [20](#)  
**U**  
unary  
    arithmetic operation, [48](#)  
    bit-wise operation, [48](#)  
unbinding  
    name, [59](#)  
UnboundLocalError, [37](#)  
unichr() (組み込み関数), [16](#)  
Unicode, [16](#)

unicode  
    object, [16](#)  
    オブジェクト, [16](#)  
unicode() (組み込み関数), [16](#), [25](#)  
Unicode Consortium, [8](#)  
UNIX, [73](#)  
unreachable object, [13](#)  
unrecognized escape sequence, [9](#)  
update() (mapping object method), [30](#)  
user-defined  
    function, [17](#)  
    function call, [47](#)  
    method, [17](#)  
    module, [62](#)  
user-defined function  
    object, [17](#), [47](#), [70](#)  
    オブジェクト, [17](#), [47](#), [70](#)  
user-defined method  
    object, [17](#)  
    オブジェクト, [17](#)

## V

value  
    default parameter, [70](#)  
value of an object, [13](#)  
ValueError  
    exception, [49](#)  
    例外, [49](#)  
values  
    writing, [55](#), [59](#)  
values() (mapping object method), [30](#)  
variable  
    free, [37](#), [59](#)

## W

while  
    実行文, [61](#), [68](#)  
    statement, [61](#), [68](#)  
whitespace, [5](#)  
writing  
    values, [55](#), [59](#)

## X

xor  
    bit-wise, [49](#)

## Y

yield

実行文, [60](#)

statement, [60](#)

#### 予約語

elif, [68](#)

else, [61](#), [68](#), [70](#)

except, [69](#)

finally, [60](#), [61](#), [70](#)

from, [62](#), [63](#)

in, [68](#)

## Z

ZeroDivisionError

exception, [48](#)

例外, [48](#)