The Python Language Reference

發Ⓕ *3.13.2*

Guido van Rossum and the Python development team

3 月 03, 2025

Contents

這份參考手⬚會描述 Python 語言的語法及「核心語意」。它雖然簡潔，但也盡量保持精確⬚完整。關於非必要的 (non-essential) ⬚建物件型⬚、⬚建函式及模組的語意，則在 library-index 中⬚明。關於此語言的非正式介紹，請參⬚ tutorial-index。對於 C 或 C++ 程式設計師，還有另外兩個手⬚：extending-index 以高階的視野⬚明如何編寫 Python 擴充模組，而 c-api-index 則詳細⬚明 C/C++ 程式設計師可用的介面。

這份參考手⬚會描述 Python 語言的語法及「核心語意」。它雖然簡潔，但也盡量保持精確⬚完整。關於非必要的 (non-essential) ⬚建物件型⬚、⬚建函式及模組的語意，則在 library-index 中⬚明。關於此語言的非正式介紹，請參⬚ tutorial-index。對於 C 或 C++ 程式設計師，還有另外兩個手⬚：extending-index 以高階的視野⬚明如何編寫 Python 擴充模組，而 c-api-index 則詳細⬚明 C/C++ 程式設計師可用的介面。

Contents

Contents

簡介

This reference manual describes the Python programming language. It is not intended as a tutorial.

While I am trying to be as precise as possible, I chose to use English rather than formal specifications for everything except syntax and lexical analysis. This should make the document more understandable to the average reader, but will leave room for ambiguities. Consequently, if you were coming from Mars and tried to re-implement Python from this document alone, you might have to guess things and in fact you would probably end up implementing quite a different language. On the other hand, if you are using Python and wonder what the precise rules about a particular area of the language are, you should definitely be able to find them here. If you would like to see a more formal definition of the language, maybe you could volunteer your time --- or invent a cloning machine :-).

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

Every Python implementation comes with a number of built-in and standard modules. These are documented in library-index. A few built-in modules are mentioned when they interact in a significant way with the language definition.

1.1 Alternate Implementations

Though there is one Python implementation which is by far the most popular, there are some alternate implementations which are of particular interest to different audiences.

Known implementations include:

CPython
    This is the original and most-maintained implementation of Python, written in C. New language features generally appear here first.

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

Python for .NET
    This implementation actually uses the CPython implementation, but is a managed .NET application and makes

.NET libraries available. It was created by Brian Lloyd. For more information, see the Python for .NET home page.

IronPython

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

PyPy

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

Each of these implementations varies in some way from the language as documented in this manual, or introduces specific information beyond what's covered in the standard Python documentation. Please refer to the implementation-specific documentation to determine what else you need to know about the specific implementation you're using.

1.2 Notation

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

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

The first line says that a `name` is an `lc_letter` followed by a sequence of zero or more `lc_letter`s and underscores. An `lc_letter` in turn is any of the single characters `'a'` through `'z'`. (This rule is actually adhered to for the names defined in lexical and grammar rules in this document.)

Each rule begins with a name (which is the name defined by the rule) and `::=`. A vertical bar (`|`) is used to separate alternatives; it is the least binding operator in this notation. A star (`*`) means zero or more repetitions of the preceding item; likewise, a plus (`+`) means one or more repetitions, and a phrase enclosed in square brackets (`[ ]`) means zero or one occurrences (in other words, the enclosed phrase is optional). The `*` and `+` operators bind as tightly as possible; parentheses are used for grouping. Literal strings are enclosed in quotes. White space is only meaningful to separate tokens. Rules are normally contained on a single line; rules with many alternatives may be formatted alternatively with each line after the first beginning with a vertical bar.

In lexical definitions (as the example above), two more conventions are used: Two literal characters separated by three dots mean a choice of any single character in the given (inclusive) range of ASCII characters. A phrase between angular brackets (`<...>`) gives an informal description of the symbol defined; e.g., this could be used to describe the notion of 'control character' if needed.

Even though the notation used is almost the same, there is a big difference between the meaning of lexical and syntactic definitions: a lexical definition operates on the individual characters of the input source, while a syntax definition operates on the stream of tokens generated by the lexical analysis. All uses of BNF in the next chapter ("Lexical Analysis") are lexical definitions; uses in subsequent chapters are syntactic definitions.

詞法分析

A Python program is read by a *parser*. Input to the parser is a stream of *tokens*, generated by the *lexical analyzer*. This chapter describes how the lexical analyzer breaks a file into tokens.

Python reads program text as Unicode code points; the encoding of a source file can be given by an encoding declaration and defaults to UTF-8, see **PEP 3120** for details. If the source file cannot be decoded, a `SyntaxError` is raised.

2.1 Line structure

A Python program is divided into a number of *logical lines*.

2.1.1 Logical lines

The end of a logical line is represented by the token NEWLINE. Statements cannot cross logical line boundaries except where NEWLINE is allowed by the syntax (e.g., between statements in compound statements). A logical line is constructed from one or more *physical lines* by following the explicit or implicit *line joining* rules.

2.1.2 Physical lines

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

When embedding Python, source code strings should be passed to Python APIs using the standard C conventions for newline characters (the `\n` character, representing ASCII LF, is the line terminator).

2.1.3 Comments

A comment starts with a hash character (#) that is not part of a string literal, and ends at the end of the physical line. A comment signifies the end of the logical line unless the implicit line joining rules are invoked. Comments are ignored by the syntax.

2.1.4 Encoding declarations

If a comment in the first or second line of the Python script matches the regular expression `coding[=:]\s*([-\w.]+)`, this comment is processed as an encoding declaration; the first group of this expression names the encoding of the source code file. The encoding declaration must appear on a line of its own. If it is the second line, the first line must also be a comment-only line. The recommended forms of an encoding expression are

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

which is recognized also by GNU Emacs, and

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

which is recognized by Bram Moolenaar's VIM.

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

If an encoding is declared, the encoding name must be recognized by Python (see standard-encodings). The encoding is used for all lexical analysis, including string literals, comments and identifiers.

2.1.5 Explicit line joining

Two or more physical lines may be joined into logical lines using backslash characters (`\`), as follows: when a physical line ends in a backslash that is not part of a string literal or comment, it is joined with the following forming a single logical line, deleting the backslash and the following end-of-line character. For example:

```python
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # 看起來像個有效日期
        return 1
```

A line ending in a backslash cannot carry a comment. A backslash does not continue a comment. A backslash does not continue a token except for string literals (i.e., tokens other than string literals cannot be split across physical lines using a backslash). A backslash is illegal elsewhere on a line outside a string literal.

2.1.6 Implicit line joining

Expressions in parentheses, square brackets or curly braces can be split over more than one physical line without using backslashes. For example:

```python
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
```

Implicitly continued lines can carry comments. The indentation of the continuation lines is not important. Blank continuation lines are allowed. There is no NEWLINE token between implicit continuation lines. Implicitly continued lines can also occur within triple-quoted strings (see below); in that case they cannot carry comments.

2.1.7 Blank lines

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

2.1.8 Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which in turn is used to determine the grouping of statements.

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

Indentation is rejected as inconsistent if a source file mixes tabs and spaces in a way that makes the meaning dependent on the worth of a tab in spaces; a `TabError` is raised in that case.

Cross-platform compatibility note: because of the nature of text editors on non-UNIX platforms, it is unwise to use a mixture of spaces and tabs for the indentation in a single source file. It should also be noted that different platforms may explicitly limit the maximum indentation level.

A formfeed character may be present at the start of the line; it will be ignored for the indentation calculations above. Formfeed characters occurring elsewhere in the leading whitespace have an undefined effect (for instance, they may reset the space count to zero).

The indentation levels of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack, as follows.

Before the first line of the file is read, a single zero is pushed on the stack; this will never be popped off again. The numbers pushed on the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, it is pushed on the stack, and one INDENT token is generated. If it is smaller, it *must* be one of the numbers occurring on the stack; all numbers on the stack that are larger are popped off, and for each number popped off a DEDENT token is generated. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

Here is an example of a correctly (though confusingly) indented piece of Python code:

```python
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
```

The following example shows various indentation errors:

```python
 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
```

(Actually, the first three errors are detected by the parser; only the last error is found by the lexical analyzer --- the indentation of `return r` does not match a level popped off the stack.)

2.1.9 Whitespace between tokens

Except at the beginning of a logical line or in string literals, the whitespace characters space, tab and formfeed can be used interchangeably to separate tokens. Whitespace is needed between two tokens only if their concatenation could otherwise be interpreted as a different token (e.g., ab is one token, but a b is two tokens).

2.2 Other tokens

Besides NEWLINE, INDENT and DEDENT, the following categories of tokens exist: *identifiers*, *keywords*, *literals*, *operators*, and *delimiters*. Whitespace characters (other than line terminators, discussed earlier) are not tokens, but serve to delimit tokens. Where ambiguity exists, a token comprises the longest possible string that forms a legal token, when read from left to right.

2.3 Identifiers and keywords

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

The syntax of identifiers in Python is based on the Unicode standard annex UAX-31, with elaboration and changes as defined below; see also **PEP 3131** for further details.

Within the ASCII range (U+0001..U+007F), the valid characters for identifiers include the uppercase and lowercase letters A through Z, the underscore _ and, except for the first character, the digits 0 through 9. Python 3.0 introduced additional characters from outside the ASCII range (see **PEP 3131**). For these characters, the classification uses the version of the Unicode Character Database as included in the unicodedata module.

Identifiers are unlimited in length. Case is significant.

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

The Unicode category codes mentioned above stand for:

- *Lu* - 大寫字母
- *Ll* - 小寫字母
- *Lt* - titlecase letters
- *Lm* - modifier letters
- *Lo* - 其他字母
- *Nl* - letter numbers
- *Mn* - nonspacing marks
- *Mc* - spacing combining marks
- *Nd* - 十進位數字
- *Pc* - connector punctuations
- *Other_ID_Start* - explicit list of characters in PropList.txt to support backwards compatibility
- *Other_ID_Continue* - likewise

All identifiers are converted into the normal form NFKC while parsing; comparison of identifiers is based on NFKC.

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

2.3.1 關鍵字

The following identifiers are used as reserved words, or *keywords* of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

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

2.3.2 軟關鍵字

在 3.10 版被加入.

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

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

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

在 3.12 版的變更: type 現在是軟關鍵字。

2.3.3 Reserved classes of identifiers

Certain classes of identifiers (besides keywords) have special meanings. These classes are identified by the patterns of leading and trailing underscore characters:

_*

    Not imported by from module import *.

_

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

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

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

> **ℹ 備Ｆ**
>
> The name _ is often used in conjunction with internationalization; refer to the documentation for the gettext module for more information on this convention.
>
> It is also commonly used for unused variables.

__*__

    System-defined names, informally known as "dunder" names. These names are defined by the interpreter and its implementation (including the standard library). Current system names are discussed in the *Special method names* section and elsewhere. More will likely be defined in future versions of Python. *Any* use of __*__ names, in any context, that does not follow explicitly documented use, is subject to breakage without warning.

__*

    Class-private names. Names in this category, when used within the context of a class definition, are re-written to use a mangled form to help avoid name clashes between "private" attributes of base and derived classes. See section *Identifiers (Names)*.

2.4 Literals

Literals are notations for constant values of some built-in types.

2.4.1 String and Bytes literals

String literals are described by the following lexical definitions:

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

bytesliteral    ::= bytesprefix(shortbytes | longbytes)
bytesprefix     ::= "b" | "B" | "br" | "Br" | "bR" | "BR" | "rb" | "rB" | "Rb" | "RB"
shortbytes      ::= "'" shortbytesitem* "'" | '"' shortbytesitem* '"'
longbytes       ::= "'''" longbytesitem* "'''" | '"""' longbytesitem* '"""'
shortbytesitem  ::= shortbyteschar | bytesescapeseq
longbytesitem   ::= longbyteschar | bytesescapeseq
shortbyteschar  ::= <any ASCII character except "\" or newline or the quote>
longbyteschar   ::= <any ASCII character except "\">
bytesescapeseq  ::= "\" <any ASCII character>
```

One syntactic restriction not indicated by these productions is that whitespace is not allowed between the *stringprefix* or *bytesprefix* and the rest of the literal. The source character set is defined by the encoding declaration; it is UTF-8 if no encoding declaration is given in the source file; see section *Encoding declarations*.

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

Bytes literals are always prefixed with 'b' or 'B'; they produce an instance of the bytes type instead of the str type. They may only contain ASCII characters; bytes with a numeric value of 128 or greater must be expressed with escapes.

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

在 3.3 版被加入: The 'rb' prefix of raw bytes literals has been added as a synonym of 'br'.

Support for the unicode legacy literal (u'value') was reintroduced to simplify the maintenance of dual Python 2.x and 3.x codebases. See **PEP 414** for more information.

A string literal with 'f' or 'F' in its prefix is a *formatted string literal*; see *f-string（f 字串）*. The 'f' may be combined with 'r', but not with 'b' or 'u', therefore raw formatted strings are possible, but formatted bytes literals are not.

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

跳脫序列

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

Escape Sequence	含義	註解
\<newline>	Backslash and newline ignored	(1)
\\	Backslash (\)	
\'	單引號 (')	
\"	雙引號 (")	
\a	ASCII Bell (BEL)	
\b	ASCII Backspace (BS)	
\f	ASCII Formfeed (FF)	
\n	ASCII Linefeed (LF)	
\r	ASCII Carriage Return (CR)	
\t	ASCII Horizontal Tab (TAB)	
\v	ASCII Vertical Tab (VT)	
ooo	Character with octal value *ooo*	(2,4)
\x*hh*	Character with hex value *hh*	(3,4)

Escape sequences only recognized in string literals are:

Escape Sequence	含義	註解
\N{*name*}	Character named *name* in the Unicode database	(5)
\u*xxxx*	Character with 16-bit hex value *xxxx*	(6)
\U*xxxxxxxx*	Character with 32-bit hex value *xxxxxxxx*	(7)

註解:

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

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

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

(2) As in Standard C, up to three octal digits are accepted.

在 3.11 版的變更: Octal escapes with value larger than `0o377` produce a `DeprecationWarning`.

在 3.12 版的變更: Octal escapes with value larger than `0o377` produce a `SyntaxWarning`. In a future Python version they will be eventually a `SyntaxError`.

(3) Unlike in Standard C, exactly two hex digits are required.

(4) In a bytes literal, hexadecimal and octal escapes denote the byte with the given value. In a string literal, these escapes denote a Unicode character with the given value.

(5) 在 3.3 版的變更: Support for name aliases[1] has been added.

(6) Exactly four hex digits are required.

(7) Any Unicode character can be encoded this way. Exactly eight hex digits are required.

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

---

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

在 3.6 版的變更: Unrecognized escape sequences produce a `DeprecationWarning`.

在 3.12 版的變更: Unrecognized escape sequences produce a `SyntaxWarning`. In a future Python version they will be eventually a `SyntaxError`.

Even in a raw literal, quotes can be escaped with a backslash, but the backslash remains in the result; for example, `r"\""` is a valid string literal consisting of two characters: a backslash and a double quote; `r"\"` is not a valid string literal (even a raw string cannot end in an odd number of backslashes). Specifically, *a raw literal cannot end in a single backslash* (since the backslash would escape the following quote character). Note also that a single backslash followed by a newline is interpreted as those two characters as part of the literal, *not* as a line continuation.

2.4.2 String literal concatenation

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

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

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

2.4.3 f-string（f 字串）

在 3.6 版被加入.

A *formatted string literal* or *f-string* is a string literal that is prefixed with `'f'` or `'F'`. These strings may contain replacement fields, which are expressions delimited by curly braces `{}`. While other string literals always have a constant value, formatted strings are really expressions evaluated at run time.

Escape sequences are decoded like in ordinary string literals (except when a literal is also marked as a raw string). After decoding, the grammar for the contents of the string is:

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

The parts of the string outside curly braces are treated literally, except that any doubled curly braces `'{{'` or `'}}'` are replaced with the corresponding single curly brace. A single opening curly bracket `'{'` marks a replacement field, which starts with a Python expression. To display both the expression text and its value after evaluation, (useful in debugging), an equal sign `'='` may be added after the expression. A conversion field, introduced by an exclamation point `'!'` may follow. A format specifier may also be appended, introduced by a colon `':'`. A replacement field ends with a closing curly bracket `'}'`.

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

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

在 3.7 版的變更: Prior to Python 3.7, an *await* expression and comprehensions containing an *async for* clause were illegal in the expressions in formatted string literals due to a problem with the implementation.

在 3.12 版的變更: Prior to Python 3.12, comments were not allowed inside f-string replacement fields.

When the equal sign `'='` is provided, the output will have the expression text, the `'='` and the evaluated value. Spaces after the opening brace `'{'`, within the expression and after the `'='` are all retained in the output. By default, the `'='` causes the `repr()` of the expression to be provided, unless there is a format specified. When a format is specified it defaults to the `str()` of the expression unless a conversion `'!r'` is declared.

在 3.8 版被加入: 等號 `'='`。

If a conversion is specified, the result of evaluating the expression is converted before formatting. Conversion `'!s'` calls `str()` on the result, `'!r'` calls `repr()`, and `'!a'` calls `ascii()`.

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

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

Formatted string literals may be concatenated, but replacement fields cannot be split across literals.

Some examples of formatted string literals:

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

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

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

在 3.12 版的變更: Prior to Python 3.12, reuse of the same quoting type of the outer f-string inside a replacement field was not possible.

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

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

在 3.12 版的變更: Prior to Python 3.12, backslashes were not permitted inside an f-string replacement field.

Formatted string literals cannot be used as docstrings, even if they do not include expressions.

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

See also **PEP 498** for the proposal that added formatted string literals, and str.format(), which uses a related format string mechanism.

2.4.4 Numeric literals

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

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

2.4.5 Integer literals

Integer literals are described by the following lexical definitions:

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

There is no limit for the length of integer literals apart from what can be stored in available memory.

Underscores are ignored for determining the numeric value of the literal. They can be used to group digits for enhanced readability. One underscore can occur between digits, and after base specifiers like 0x.

Note that leading zeros in a non-zero decimal number are not allowed. This is for disambiguation with C-style octal literals, which Python used before version 3.0.

Some examples of integer literals:

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

在 3.6 版的變更: Underscores are now allowed for grouping purposes in literals.

2.4.6 浮點數常數 (Floating-point literals)

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

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

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

一些浮點數常數的範例:

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

在 3.6 版的變更: Underscores are now allowed for grouping purposes in literals.

2.4.7 Imaginary literals

Imaginary literals are described by the following lexical definitions:

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

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

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

2.5 Operators

The following tokens are operators:

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

2.6 Delimiters

The following tokens serve as delimiters in the grammar:

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

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

The following printing ASCII characters have special meaning as part of other tokens or are otherwise significant to the lexical analyzer:

```
'          "          #          \
```

The following printing ASCII characters are not used in Python. Their occurrence outside string literals and comments is an unconditional error:

```
$          ?               `
```

⬚解

資料模型

3.1 物件、數值和型別

物件是 Python 裡資料的抽象表示方式。一個 Python 程式當中的所有資料皆由物件或物件之間的關係來呈現。（某種意義上，同時符合 Von Neumann 對於「儲存程式型電腦」（"stored program computer"）的模型，程式碼也是以物件呈現的。）

每個物件都有一個識別性、型別，和數值。物件的識別性在物件建立後永遠不會改變；你也可以把它想成是該物件在記憶體中的位址。 *is* 運算子會比較兩個物件的識別性是否相同；id() 函式則會回傳代表一個該物件的識別性的整數。

在 CPython 當中，id(x) 就是 x 所儲存在的記憶體位址。

一個物件的型別定了該物件所支援的操作（例如「它有長度嗎?」），也同時定義該型別的物件能夠擁有的數值。type() 函式會回傳一個物件的型別（而該型別本身也是一個物件）。如同它的識別性，一個物件的型別 (*type*) 也是不可變的。[1]

某些物件的數值可被改變，這種物件稱作「可變的」(*mutable*)；建立後數值不能變更的物件則稱作「不可變的」(*immutable*)。（不可變的容器物件中如果包含對於可變物件的參照，則後者的數值改變的時候前者的數值也會跟著一起改變；這種時候該容器仍會被當成是不可變的，因為它包含的物件集合仍然無法變更。因此可變或不可變嚴格說起來不等同於數值是否能被改變，它的定義有其他不明顯的細節。）一個物件是否可變取決於它的型別；舉例來說，數字、字串和 tuple 是不可變的，而字典與串列則是可變的。

物件永遠不會被明示的摧毀；但當它們變得不再能被存取的時候可能會被作為垃圾回收。每個實作都能延後垃圾回收或是乾脆忽略它 --- 垃圾回收如何進行完全取決於各個實作，只要沒有被回收的物件仍是可達的。

CPython 目前使用一種參照計數的方案，並提供可選的循環連結垃圾延遲偵測，這個方案會在大部分物件變得不可存取時馬上回收它們，但不保證能夠回收包含循環參照的垃圾。關於控制循環垃圾回收的資訊請見 gc 模組的說明文件。其他實作的行為不會相同，CPython 也有可能改變，因此請不要仰賴物件在變得不可存取時能夠馬上被最終化（亦即你應該總是明確關閉檔案）。

請注意，使用一個實作的追蹤或除錯工具可能會讓原本能夠回收的物件被維持存活。也請注意，使用 *try...except* 陳述式來抓捕例外也可能會讓物件維持存活。

某些物件包含對於「外部」資源的參照，像是開啟的檔案或是視窗。基本上這些資源會在物件被回收時釋放，但因為垃圾回收不保證會發生，這種物件也會提供明確釋放外部資源的方式 --- 通常是 close()

---

[1] It *is* possible in some cases to change an object's type, under certain controlled conditions. It generally isn't a good idea though, since it can lead to some very strange behaviour if it is handled incorrectly.

method。強烈建議各個程式明確關閉這種物件。*try...finally* 陳述式與 *with* 陳述式提供進行明確關閉的方便手段。

某些物件包含對於其他物件的參照；這種物件被叫做「容器」。容器的範例有 tuple、串列與字典。這些參照是容器的數值的一部分。通常當我們提到容器的數值的時候，我們指的是其中包含的物件的數值，而不是它們的識別性；但當我們提到容器是否可變的時候，我們指的是直接包含在其中的物件的識別性。因此，如果一個不可變的容器（像一個 tuple）包含對於可變物件的參照，該可變物件被變更時該容器的數值也會跟著變更。

型別幾乎影響物件行為的所有面向。就連物件識別性的重要性某種程度上也受型別影響：對於不可變的型別，計算新數值的操作可能其實會回傳一個某個相同型別且相同數值的現存物件的參照；對於可變型別這則不會發生。舉例來說，在進行 a = 1; b = 1 之後，*a* 和 *b* 可能會參照同一個物件，也可能不會，取決於所使用的實作。這是因為 int 是不可變的型別，因此 1 的參照可以重新利用。這個行為取決於所使用的實作，因此不應該依賴它，但在進行物件識別性測試的時候還是需要注意有這件事情。而在進行 c = []; d = [] 之後，*c* 和 *d* 則保證會參照兩個不同、獨特、且新建立的空白串列。（請注意，e = f = [] 則會將同一個物件同時指派給 *e* 和 *f*。）

3.2 標準型別階層

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

Some of the type descriptions below contain a paragraph listing 'special attributes.' These are attributes that provide access to the implementation and are not intended for general use. Their definition may change in the future.

3.2.1 None

這個型別只有一個數值。只有一個物件有這個數值。這個物件由內建名稱 None 存取。它用來在許多情況下代表數值不存在，例如沒有明確回傳任何東西的函式就會回傳這個物件。它的真值是 false。

3.2.2 NotImplemented

這個型別只有一個數值。只有一個物件有這個數值。這個物件由內建名稱 NotImplemented 存取。數字方法和 rich comparison 方法應該在沒有由所提供的運算元實作該操作的時候回傳這個數值。（直譯器接下來則會依運算子嘗試反轉的操作或是其他的後備方案。）它不應該在預期布林值的情境中被計算。

更多細節請見 implementing-the-arithmetic-operations。

在 3.9 版的變更: 在預期布林值的情境中計算 NotImplemented 的行為已棄用。雖然目前它會計算為 true，它也會同時發出 DeprecationWarning。從某個未來 Python 版本開始這將會引發 TypeError。

3.2.3 Ellipsis

這個型別只有一個數值。只有一個物件有這個數值。這個物件由文本 ... 或內建名稱 Ellipsis 存取。它的真值是 true。

3.2.4 `numbers.Number`

These are created by numeric literals and returned as results by arithmetic operators and arithmetic built-in functions. Numeric objects are immutable; once created their value never changes. Python numbers are of course strongly related to mathematical numbers, but subject to the limitations of numerical representation in computers.

The string representations of the numeric classes, computed by `__repr__()` and `__str__()`, have the following properties:

- They are valid numeric literals which, when passed to their class constructor, produce an object having the value of the original numeric.

- The representation is in base 10, when possible.

- Leading zeros, possibly excepting a single zero before a decimal point, are not shown.

- Trailing zeros, possibly excepting a single zero after a decimal point, are not shown.

- A sign is shown only when the number is negative.

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

`numbers.Integral`

These represent elements from the mathematical set of integers (positive and negative).

> ℹ️ **備⬚**
>
> The rules for integer representation are intended to give the most meaningful interpretation of shift and mask operations involving negative integers.

There are two types of integers:

Integers (`int`)
: These represent numbers in an unlimited range, subject to available (virtual) memory only. For the purpose of shift and mask operations, a binary representation is assumed, and negative numbers are represented in a variant of 2's complement which gives the illusion of an infinite string of sign bits extending to the left.

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

`numbers.Real (float)`

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

`numbers.Complex (complex)`

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

3.2.5 Sequences

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

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

Some sequences also support "extended slicing" with a third "step" parameter: `a[i:j:k]` selects all items of a with index x where `x = i + n*k`, $n >= 0$ and $i <= x < j$.

Sequences are distinguished according to their mutability:

Immutable sequences

An object of an immutable sequence type cannot change once it is created. (If the object contains references to other objects, these other objects may be mutable and may be changed; however, the collection of objects directly referenced by an immutable object cannot change.)

The following types are immutable sequences:

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

Tuple（元組）
> The items of a tuple are arbitrary Python objects. Tuples of two or more items are formed by comma-separated lists of expressions. A tuple of one item (a 'singleton') can be formed by affixing a comma to an expression (an expression by itself does not create a tuple, since parentheses must be usable for grouping of expressions). An empty tuple can be formed by an empty pair of parentheses.

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

可變序列

Mutable sequences can be changed after they are created. The subscription and slicing notations can be used as the target of assignment and *`del`* (delete) statements.

> ⓘ **備⊞**
>
> The `collections` and `array` module provide additional examples of mutable sequence types.

There are currently two intrinsic mutable sequence types:

List（串列）
> The items of a list are arbitrary Python objects. Lists are formed by placing a comma-separated list of expressions in square brackets. (Note that there are no special cases needed to form lists of length 0 or 1.)

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

3.2.6 Set（集合）型⊞

These represent unordered, finite sets of unique, immutable objects. As such, they cannot be indexed by any subscript. However, they can be iterated over, and the built-in function `len()` returns the number of items in a set. Common uses for sets are fast membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

For set elements, the same immutability rules apply as for dictionary keys. Note that numeric types obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`), only one of them can be contained in a set.

There are currently two intrinsic set types:

Set（集合）

These represent a mutable set. They are created by the built-in `set()` constructor and can be modified afterwards by several methods, such as `add()`.

Frozen set（凍結集合）

These represent an immutable set. They are created by the built-in `frozenset()` constructor. As a frozenset is immutable and *hashable*, it can be used again as an element of another set, or as a dictionary key.

3.2.7 對映

These represent finite sets of objects indexed by arbitrary index sets. The subscript notation `a[k]` selects the item indexed by `k` from the mapping `a`; this can be used in expressions and as the target of assignments or *del* statements. The built-in function `len()` returns the number of items in a mapping.

There is currently a single intrinsic mapping type:

字典

These represent finite sets of objects indexed by nearly arbitrary values. The only types of values not acceptable as keys are values containing lists or dictionaries or other mutable types that are compared by value rather than by object identity, the reason being that the efficient implementation of dictionaries requires a key's hash value to remain constant. Numeric types used for keys obey the normal rules for numeric comparison: if two numbers compare equal (e.g., `1` and `1.0`) then they can be used interchangeably to index the same dictionary entry.

Dictionaries preserve insertion order, meaning that keys will be produced in the same order they were added sequentially over the dictionary. Replacing an existing key does not change the order, however removing a key and re-inserting it will add it to the end instead of keeping its old place.

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

The extension modules `dbm.ndbm` and `dbm.gnu` provide additional examples of mapping types, as does the `collections` module.

在 3.7 版的變更: Dictionaries did not preserve insertion order in versions of Python before 3.6. In CPython 3.6, insertion order was preserved, but it was considered an implementation detail at that time rather than a language guarantee.

3.2.8 可呼叫型⊞

These are the types to which the function call operation (see section *Calls*) can be applied:

自訂函式

A user-defined function object is created by a function definition (see section 函式定義). It should be called with an argument list containing the same number of items as the function's formal parameter list.

特殊唯讀屬性

屬性	含義
function.**__globals__**	A reference to the `dictionary` that holds the function's *global variables* -- the global namespace of the module in which the function was defined.
function.**__closure__**	`None` or a `tuple` of cells that contain bindings for the names specified in the *co_freevars* attribute of the function's *code object*. A cell object has the attribute `cell_contents`. This can be used to get the value of the cell, as well as set the value.

特殊可寫屬性

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

屬性	含義
function.**__doc__**	函式的文件字串，若不可用則⽥ None。
function.**__name__**	The function's name. See also: __name__ attributes.
function.**__qualname__**	The function's *qualified name*. See also: __qualname__ attributes. 在 3.3 版被加入.
function.**__module__**	The name of the module the function was defined in, or None if unavailable.
function.**__defaults__**	A tuple containing default *parameter* values for those parameters that have defaults, or None if no parameters have a default value.
function.**__code__**	代表編譯函式主體的程式碼物件。
function.**__dict__**	The namespace supporting arbitrary function attributes. See also: *__dict__ attributes*.
function.**__annotations__**	A dictionary containing annotations of *parameters*. The keys of the dictionary are the parameter names, and 'return' for the return annotation, if provided. See also: annotations-howto.
function.**__kwdefaults__**	A dictionary containing defaults for keyword-only *parameters*.
function.**__type_params__**	A tuple containing the *type parameters* of a *generic function*. 在 3.12 版被加入.

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

CPython 實作細節： CPython's current implementation only supports function attributes on user-defined functions. Function attributes on *built-in functions* may be supported in the future.

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

實例方法

An instance method object combines a class, a class instance and any callable object (normally a user-defined function).

特殊唯讀屬性：

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

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

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

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

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

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

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

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

Ⓕ生器函式

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

Coroutine（協程）函式

A function or method which is defined using *async def* is called a *coroutine function*. Such a function, when called, returns a *coroutine* object. It may contain *await* expressions, as well as *async with* and *async for* statements. See also the 協程物件 section.

非同步Ⓕ生器函式

A function or method which is defined using *async def* and which uses the *yield* statement is called a *asynchronous generator function*. Such a function, when called, returns an *asynchronous iterator* object which can be used in an *async for* statement to execute the body of the function.

Calling the asynchronous iterator's *aiterator.__anext__* method will return an *awaitable* which when awaited will execute until it provides a value using the *yield* expression. When the function executes an empty *return*

statement or falls off the end, a `StopAsyncIteration` exception is raised and the asynchronous iterator will have reached the end of the set of values to be yielded.

⊞建函式

一個⊞建函式物件是一個 C 函式的 wrapper。⊞建函式的範例有 `len()` 和 `math.sin()` (`math` 是一個標準的⊞建模組)。⊞建函式的引數數量與其型⊞由其包裝的 C 函式所⊞定。特殊唯讀屬性：

- `__doc__` 是函式的文件字串，若不可用則⊞ `None`。請見 *function.__doc__*。
- `__name__` 是函式的名稱。請見 *function.__name__*。
- `__self__` is set to `None` (but see the next item).
- `__module__` is the name of the module the function was defined in or `None` if unavailable. See *function.__module__*.

⊞建方法

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

Classes

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

類⊞實例

Instances of arbitrary classes can be made callable by defining a *__call__()* method in their class.

3.2.9 模組

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

Attribute assignment updates the module's namespace dictionary, e.g., `m.x = 1` is equivalent to `m.__dict__["x"] = 1`.

Import-related attributes on module objects

Module objects have the following attributes that relate to the *import system*. When a module is created using the machinery associated with the import system, these attributes are filled in based on the module's *spec*, before the *loader* executes and loads the module.

To create a module dynamically rather than using the import system, it's recommended to use `importlib.util.module_from_spec()`, which will set the various import-controlled attributes to appropriate values. It's also possible to use the `types.ModuleType` constructor to create modules directly, but this technique is more error-prone, as most attributes must be manually set on the module object after it has been created when using this approach.

> ☢ 警示
>
> With the exception of *__name__*, it is **strongly** recommended that you rely on *__spec__* and its attributes instead of any of the other individual attributes listed in this subsection. Note that updating an attribute on `__spec__` will not update the corresponding attribute on the module itself:

```
>>> import typing
>>> typing.__name__, typing.__spec__.name
('typing', 'typing')
>>> typing.__spec__.name = 'spelling'
>>> typing.__name__, typing.__spec__.name
('typing', 'spelling')
>>> typing.__name__ = 'keyboard_smashing'
>>> typing.__name__, typing.__spec__.name
('keyboard_smashing', 'spelling')
```

module.**__name__**

> The name used to uniquely identify the module in the import system. For a directly executed module, this will be set to `"__main__"`.
>
> This attribute must be set to the fully qualified name of the module. It is expected to match the value of `module.__spec__.name`.

module.**__spec__**

> A record of the module's import-system-related state.
>
> Set to the `module spec` that was used when importing the module. See 模組規格 for more details.
>
> 在 3.4 版被加入.

module.**__package__**

> The *package* a module belongs to.
>
> If the module is top-level (that is, not a part of any specific package) then the attribute should be set to `''` (the empty string). Otherwise, it should be set to the name of the module's package (which can be equal to *module.__name__* if the module itself is a package). See **PEP 366** for further details.
>
> This attribute is used instead of *__name__* to calculate explicit relative imports for main modules. It defaults to `None` for modules created dynamically using the `types.ModuleType` constructor; use `importlib.util.module_from_spec()` instead to ensure the attribute is set to a `str`.
>
> It is **strongly** recommended that you use `module.__spec__.parent` instead of `module.__package__`. *__package__* is now only used as a fallback if `__spec__.parent` is not set, and this fallback path is deprecated.
>
> 在 3.4 版的變更: This attribute now defaults to `None` for modules created dynamically using the `types.ModuleType` constructor. Previously the attribute was optional.
>
> 在 3.6 版的變更: The value of `__package__` is expected to be the same as `__spec__.parent`. *__package__* is now only used as a fallback during import resolution if `__spec__.parent` is not defined.
>
> 在 3.10 版的變更: `ImportWarning` is raised if an import resolution falls back to `__package__` instead of `__spec__.parent`.
>
> 在 3.12 版的變更: Raise `DeprecationWarning` instead of `ImportWarning` when falling back to `__package__` during import resolution.
>
> Deprecated since version 3.13, will be removed in version 3.15: `__package__` will cease to be set or taken into consideration by the import system or standard library.

module.**__loader__**

> The *loader* object that the import machinery used to load the module.
>
> This attribute is mostly useful for introspection, but can be used for additional loader-specific functionality, for example getting data associated with a loader.
>
> `__loader__` defaults to `None` for modules created dynamically using the `types.ModuleType` constructor; use `importlib.util.module_from_spec()` instead to ensure the attribute is set to a *loader* object.
>
> It is **strongly** recommended that you use `module.__spec__.loader` instead of `module.__loader__`.

在 3.4 版的變更: This attribute now defaults to `None` for modules created dynamically using the `types.ModuleType` constructor. Previously the attribute was optional.

Deprecated since version 3.12, will be removed in version 3.16: 在模組上設定 `__loader__` 而非有設定 `__spec__.loader` 的做法已被棄用。在 Python 3.16 中，引入系統或標准函式庫將不再設定或考慮 `__loader__`。

module.**`__path__`**

A (possibly empty) *sequence* of strings enumerating the locations where the package's submodules will be found. Non-package modules should not have a `__path__` attribute. See *__path__ attributes on modules* for more details.

It is **strongly** recommended that you use `module.__spec__.submodule_search_locations` instead of `module.__path__`.

module.**`__file__`**

module.**`__cached__`**

`__file__` and `__cached__` are both optional attributes that may or may not be set. Both attributes should be a `str` when they are available.

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

If `__file__` is set then the `__cached__` attribute might also be set, which is the path to any compiled version of the code (for example, a byte-compiled file). The file does not need to exist to set this attribute; the path can simply point to where the compiled file *would* exist (see **PEP 3147**).

Note that `__cached__` may be set even if `__file__` is not set. However, that scenario is quite atypical. Ultimately, the *loader* is what makes use of the module spec provided by the *finder* (from which `__file__` and `__cached__` are derived). So if a loader can load from a cached module but otherwise does not load from a file, that atypical scenario may be appropriate.

It is **strongly** recommended that you use `module.__spec__.cached` instead of `module.__cached__`.

Deprecated since version 3.13, will be removed in version 3.15: Setting `__cached__` on a module while failing to set `__spec__.cached` is deprecated. In Python 3.15, `__cached__` will cease to be set or taken into consideration by the import system or standard library.

模組物件的其他可寫入屬性

As well as the import-related attributes listed above, module objects also have the following writable attributes:

module.**`__doc__`**
模組的文件字串，若不可用則為 `None`。請見 `__doc__` attributes。

module.**`__annotations__`**
A dictionary containing *variable annotations* collected during module body execution. For best practices on working with `__annotations__`, please see annotations-howto.

模組字典

Module objects also have the following special read-only attribute:

module.**`__dict__`**

The module's namespace as a dictionary object. Uniquely among the attributes listed here, `__dict__` cannot be accessed as a global variable from within a module; it can only be accessed as an attribute on module objects.

CPython 實作細節： Because of the way CPython clears module dictionaries, the module dictionary will be cleared when the module falls out of scope even if the dictionary still has live references. To avoid this, copy the dictionary or keep the module around while using its dictionary directly.

3.2.10 Custom classes

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

When a class attribute reference (for class `C`, say) would yield a class method object, it is transformed into an instance method object whose *__self__* attribute is `C`. When it would yield a `staticmethod` object, it is transformed into the object wrapped by the static method object. See section 實作描述器 for another way in which attributes retrieved from a class may differ from those actually contained in its *__dict__*.

Class attribute assignments update the class's dictionary, never the dictionary of a base class.

A class object can be called (see above) to yield a class instance (see below).

特殊屬性

屬性	含義
type.**__name__**	The class's name. See also: __name__ attributes.
type.**__qualname__**	The class's *qualified name*. See also: __qualname__ attributes.
type.**__module__**	The name of the module in which the class was defined.
type.**__dict__**	A mapping proxy providing a read-only view of the class's namespace. See also: *__dict__ attributes*.
type.**__bases__**	A tuple containing the class's bases. In most cases, for a class defined as class X(A, B, C), X.__bases__ will be exactly equal to (A, B, C).
type.**__doc__**	The class's documentation string, or None if undefined. Not inherited by subclasses.
type.**__annotations__**	A dictionary containing *variable annotations* collected during class body execution. For best practices on working with __annotations__, please see annotations-howto. ☢ **警示** Accessing the __annotations__ attribute of a class object directly may yield incorrect results in the presence of metaclasses. In addition, the attribute may not exist for some classes. Use inspect.get_annotations() to retrieve class annotations safely.
type.**__type_params__**	A tuple containing the *type parameters* of a *generic class*. 在 3.12 版被加入.
type.**__static_attributes__**	A tuple containing names of attributes of this class which are assigned through self.X from any function in its body. 在 3.13 版被加入.
type.**__firstlineno__**	The line number of the first line of the class definition, including decorators. Setting the __module__ attribute removes the __firstlineno__ item from the type's dictionary. 在 3.13 版被加入.
type.**__mro__**	The tuple of classes that are considered when looking for base classes during method resolution.

特殊方法

In addition to the special attributes described above, all Python classes also have the following two methods available:

type.**mro**()

> This method can be overridden by a metaclass to customize the method resolution order for its instances. It is

called at class instantiation, and its result is stored in *__mro__*.

type.**__subclasses__**()

> Each class keeps a list of weak references to its immediate subclasses. This method returns a list of all those references still alive. The list is in definition order. Example:

```
>>> class A: pass
>>> class B(A): pass
>>> A.__subclasses__()
[<class 'B'>]
```

3.2.11 類⒡實例

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

Attribute assignments and deletions update the instance's dictionary, never a class's dictionary. If the class has a *__setattr__()* or *__delattr__()* method, this is called instead of updating the instance dictionary directly.

Class instances can pretend to be numbers, sequences, or mappings if they have methods with certain special names. See section *Special method names*.

特殊屬性

object.**__class__**

> The class to which a class instance belongs.

object.**__dict__**

> A dictionary or other mapping object used to store an object's (writable) attributes. Not all instances have a __dict__ attribute; see the section on *__slots__* for more details.

3.2.12 I/O objects (also known as file objects)

A *file object* represents an open file. Various shortcuts are available to create file objects: the open() built-in function, and also os.popen(), os.fdopen(), and the makefile() method of socket objects (and perhaps by other functions or methods provided by extension modules).

The objects sys.stdin, sys.stdout and sys.stderr are initialized to file objects corresponding to the interpreter's standard input, output and error streams; they are all open in text mode and therefore follow the interface defined by the io.TextIOBase abstract class.

3.2.13 Internal types

A few types used internally by the interpreter are exposed to the user. Their definitions may change with future versions of the interpreter, but they are mentioned here for completeness.

程式碼物件

Code objects represent *byte-compiled* executable Python code, or *bytecode*. The difference between a code object and a function object is that the function object contains an explicit reference to the function's globals (the module in which it was defined), while a code object contains no context; also the default argument values are stored in the function object, not in the code object (because they represent values calculated at run-time). Unlike function objects, code objects are immutable and contain no references (directly or indirectly) to mutable objects.

特殊唯讀屬性

codeobject.**co_name**	函式名稱
codeobject.**co_qualname**	The fully qualified function name 在 3.11 版被加入.
codeobject.**co_argcount**	The total number of positional *parameters* (including positional-only parameters and parameters with default values) that the function has
codeobject.**co_posonlyargcount**	The number of positional-only *parameters* (including arguments with default values) that the function has
codeobject.**co_kwonlyargcount**	The number of keyword-only *parameters* (including arguments with default values) that the function has
codeobject.**co_nlocals**	The number of *local variables* used by the function (including parameters)
codeobject.**co_varnames**	A `tuple` containing the names of the local variables in the function (starting with the parameter names)
codeobject.**co_cellvars**	A `tuple` containing the names of *local variables* that are referenced from at least one *nested scope* inside the function
codeobject.**co_freevars**	A `tuple` containing the names of *free (closure) variables* that a *nested scope* references in an outer scope. See also `function.__closure__`. Note: references to global and builtin names are *not* included.
codeobject.**co_code**	A string representing the sequence of *bytecode* instructions in the function
codeobject.**co_consts**	A `tuple` containing the literals used by the *bytecode* in the function
codeobject.**co_names**	A `tuple` containing the names used by the *bytecode* in the function
codeobject.**co_filename**	The name of the file from which the code was compiled
codeobject.**co_firstlineno**	The line number of the first line of the function
codeobject.**co_lnotab**	A string encoding the mapping from *bytecode* offsets to line numbers. For details, see the source code of the interpreter. 在 3.12 版之後被棄用: This attribute of code objects is deprecated, and may be removed in Python 3.15.
codeobject.**co_stacksize**	The required stack size of the code object
codeobject.**co_flags**	An `integer` encoding a number of flags for the interpreter.

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

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

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

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

用於程式碼物件的方法

codeobject.**co_positions**()

> Returns an iterable over the source code positions of each *bytecode* instruction in the code object.
>
> The iterator returns `tuple`s containing the (`start_line, end_line, start_column, end_column`). The *i-th* tuple corresponds to the position of the source code that compiled to the *i-th* code unit. Column information is 0-indexed utf-8 byte offsets on the given source line.
>
> This positional information can be missing. A non-exhaustive lists of cases where this may happen:
>
> - Running the interpreter with `-X no_debug_ranges`.
> - Loading a pyc file compiled while using `-X no_debug_ranges`.
> - Position tuples corresponding to artificial instructions.
> - Line and column numbers that can't be represented due to implementation specific limitations.
>
> When this occurs, some or all of the tuple elements can be `None`.
>
> 在 3.11 版被加入.
>
> > **🛈 備[F]**
> >
> > This feature requires storing column positions in code objects which may result in a small increase of disk usage of compiled Python files or interpreter memory usage. To avoid storing the extra information and/or deactivate printing the extra traceback information, the `-X no_debug_ranges` command line flag or the `PYTHONNODEBUGRANGES` environment variable can be used.

codeobject.**co_lines**()

> Returns an iterator that yields information about successive ranges of *bytecode*s. Each item yielded is a (`start, end, lineno`) tuple:
>
> - `start` (an `int`) represents the offset (inclusive) of the start of the *bytecode* range
> - `end` (an `int`) represents the offset (exclusive) of the end of the *bytecode* range
> - `lineno` is an `int` representing the line number of the *bytecode* range, or `None` if the bytecodes in the given range have no line number
>
> The items yielded will have the following properties:
>
> - The first range yielded will have a `start` of 0.
> - The (`start, end`) ranges will be non-decreasing and consecutive. That is, for any pair of `tuple`s, the `start` of the second will be equal to the `end` of the first.
> - No range will be backwards: `end >= start` for all triples.
> - The last `tuple` yielded will have `end` equal to the size of the *bytecode*.

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

在 3.10 版被加入.

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

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

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

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

在 3.8 版被加入.

Frame objects

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

特殊唯讀屬性

frame.**f_back**	Points to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame
frame.**f_code**	在這個 frame 中執行的*程式碼物件 (code object)*。存取這個屬性會引發一個附帶引數 `obj` 與 `"f_code"` 的稽核事件 `object.__getattr__`。
frame.**f_locals**	The mapping used by the frame to look up *local variables*. If the frame refers to an *optimized scope*, this may return a write-through proxy object. 在 3.13 版的變更: Return a proxy for optimized scopes.
frame.**f_globals**	The dictionary used by the frame to look up *global variables*
frame.**f_builtins**	The dictionary used by the frame to look up *built-in (intrinsic) names*
frame.**f_lasti**	The "precise instruction" of the frame object (this is an index into the *bytecode* string of the *code object*)

特殊可寫屬性

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

Frame object methods

Frame objects support one method:

frame.**clear**()

> This method clears all references to *local variables* held by the frame. Also, if the frame belonged to a *generator*, the generator is finalized. This helps break reference cycles involving frame objects (for example when catching an exception and storing its *traceback* for later use).
>
> `RuntimeError` is raised if the frame is currently executing or suspended.
>
> 在 3.4 版被加入.
>
> 在 3.13 版的變更: Attempting to clear a suspended frame raises `RuntimeError` (as has always been the case for executing frames).

Traceback objects

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

在 3.7 版的變更: Traceback objects can now be explicitly instantiated from Python code.

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

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

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

特殊唯讀屬性：

traceback.**tb_frame**	Points to the execution *frame* of the current level. 存 取 此 屬 性 會 引 發 ⼀ 個 附 帶 引 數 obj 與 "tb_frame" 的稽核事件 object.__getattr__。
traceback.**tb_lineno**	Gives the line number where the exception occurred
traceback.**tb_lasti**	Indicates the "precise instruction".

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

traceback.**tb_next**

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

> 在 3.7 版的變更: This attribute is now writable

Slice objects

Slice objects are used to represent slices for *__getitem__()* methods. They are also created by the built-in slice() function.

Special read-only attributes: start is the lower bound; stop is the upper bound; step is the step value; each is None if omitted. These attributes can have any type.

Slice objects support one method:

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

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

Static method objects

Static method objects provide a way of defeating the transformation of function objects to method objects described above. A static method object is a wrapper around any other object, usually a user-defined method object. When a static method object is retrieved from a class or a class instance, the object actually returned is the wrapped object, which is not subject to any further transformation. Static method objects are also callable. Static method objects are created by the built-in staticmethod() constructor.

Class method objects

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

3.3 Special method names

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

Setting a special method to `None` indicates that the corresponding operation is not available. For example, if a class sets *__iter__()* to `None`, the class is not iterable, so calling `iter()` on its instances will raise a `TypeError` (without falling back to *__getitem__()*).[2]

When implementing a class that emulates any built-in type, it is important that the emulation only be implemented to the degree that it makes sense for the object being modelled. For example, some sequences may work well with retrieval of individual elements, but extracting a slice may not make sense. (One example of this is the `NodeList` interface in the W3C's Document Object Model.)

3.3.1 Basic customization

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

> Called to create a new instance of class *cls*. *__new__()* is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of *__new__()* should be the new object instance (usually an instance of *cls*).
>
> Typical implementations create a new instance of the class by invoking the superclass's *__new__()* method using `super().__new__(cls[, ...])` with appropriate arguments and then modifying the newly created instance as necessary before returning it.
>
> If *__new__()* is invoked during object construction and it returns an instance of *cls*, then the new instance's *__init__()* method will be invoked like `__init__(self[, ...])`, where *self* is the new instance and the remaining arguments are the same as were passed to the object constructor.
>
> If *__new__()* does not return an instance of *cls*, then the new instance's *__init__()* method will not be invoked.
>
> *__new__()* is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation. It is also commonly overridden in custom metaclasses in order to customize class creation.

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

> Called after the instance has been created (by *__new__()*), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an *__init__()* method, the derived class's *__init__()* method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: `super().__init__([args...])`.
>
> Because *__new__()* and *__init__()* work together in constructing objects (*__new__()* to create it, and *__init__()* to customize it), no non-`None` value may be returned by *__init__()*; doing so will cause a `TypeError` to be raised at runtime.

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

> Called when the instance is about to be destroyed. This is also called a finalizer or (improperly) a destructor. If a base class has a *__del__()* method, the derived class's *__del__()* method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.
>
> It is possible (though not recommended!) for the *__del__()* method to postpone destruction of the instance by creating a new reference to it. This is called object *resurrection*. It is implementation-dependent whether *__del__()* is called a second time when a resurrected object is about to be destroyed; the current *CPython* implementation only calls it once.
>
> It is not guaranteed that *__del__()* methods are called for objects that still exist when the interpreter exits. `weakref.finalize` provides a straightforward way to register a cleanup function to be called when an object is garbage collected.

> ---
> **ℹ 備 $\mathbb{F}$**
>
> `del x` doesn't directly call `x.__del__()` --- the former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero.
> ---

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

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

> ➡ **也參考**
>
> Documentation for the `gc` module.

> ⚠ **警告**
>
> Due to the precarious circumstances under which `__del__()` methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to `sys.stderr` instead. In particular:
>
> - `__del__()` can be invoked when arbitrary code is being executed, including from any arbitrary thread. If `__del__()` needs to take a lock or invoke any other blocking resource, it may deadlock as the resource may already be taken by the code that gets interrupted to execute `__del__()`.
>
> - `__del__()` can be executed during interpreter shutdown. As a consequence, the global variables it needs to access (including other modules) may already have been deleted or set to `None`. Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the `__del__()` method is called.

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

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

This is typically used for debugging, so it is important that the representation is information-rich and unambiguous. A default implementation is provided by the `object` class itself.

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

Called by `str(object)`, the default `__format__()` implementation, and the built-in function `print()`, to compute the "informal" or nicely printable string representation of an object. The return value must be a str object.

This method differs from `object.__repr__()` in that there is no expectation that `__str__()` return a valid Python expression: a more convenient or concise representation can be used.

The default implementation defined by the built-in type `object` calls `object.__repr__()`.

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

Called by bytes to compute a byte-string representation of an object. This should return a `bytes` object. The `object` class itself does not provide this method.

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

Called by the `format()` built-in function, and by extension, evaluation of *formatted string literals* and the `str.format()` method, to produce a "formatted" string representation of an object. The *format_spec* argument is a string that contains a description of the formatting options desired. The interpretation of the *format_spec* argument is up to the type implementing `__format__()`, however most classes will either delegate formatting to one of the built-in types, or use a similar formatting option syntax.

See formatspec for a description of the standard formatting syntax.

回傳值必須是個字串物件。

The default implementation by the `object` class should be given an empty *format_spec* string. It delegates to `__str__()`.

在 3.4 版的變更: The __format__ method of `object` itself raises a `TypeError` if passed any non-empty string.

在 3.7 版 的 變 更: `object.__format__(x, '')` is now equivalent to `str(x)` rather than `format(str(x), '')`.

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

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

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

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

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

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

These are the so-called "rich comparison" methods. The correspondence between operator symbols and method names is as follows: x<y calls `x.__lt__(y)`, x<=y calls `x.__le__(y)`, x==y calls `x.__eq__(y)`, x!=y calls `x.__ne__(y)`, x>y calls `x.__gt__(y)`, and x>=y calls `x.__ge__(y)`.

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

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

By default, the `object` class provides implementations consistent with *Value comparisons*: equality compares according to object identity, and order comparisons raise `TypeError`. Each default method may generate these results directly, but may also return `NotImplemented`.

See the paragraph on *__hash__()* for some important notes on creating *hashable* objects which support custom comparison operations and are usable as dictionary keys.

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

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

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

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

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

> **ⓘ 備註**
>
> hash() truncates the value returned from an object's custom *__hash__()* method to the size of a Py_ssize_t. This is typically 8 bytes on 64-bit builds and 4 bytes on 32-bit builds. If an object's *__hash__()* must interoperate on builds of different bit sizes, be sure to check the width on all supported builds. An easy way to do this is with python -c "import sys; print(sys.hash_info. width)".

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

User-defined classes have *__eq__()* and *__hash__()* methods by default (inherited from the object class); with them, all objects compare unequal (except with themselves) and x.__hash__() returns an appropriate value such that x == y implies both that x is y and hash(x) == hash(y).

A class that overrides *__eq__()* and does not define *__hash__()* will have its *__hash__()* implicitly set to None. When the *__hash__()* method of a class is None, instances of the class will raise an appropriate TypeError when a program attempts to retrieve their hash value, and will also be correctly identified as unhashable when checking isinstance(obj, collections.abc.Hashable).

If a class that overrides *__eq__()* needs to retain the implementation of *__hash__()* from a parent class, the interpreter must be told this explicitly by setting __hash__ = <ParentClass>.__hash__.

If a class that does not override *__eq__()* wishes to suppress hash support, it should include __hash__ = None in the class definition. A class which defines its own *__hash__()* that explicitly raises a TypeError would be incorrectly identified as hashable by an isinstance(obj, collections.abc.Hashable) call.

> **ⓘ 備註**
>
> By default, the *__hash__()* values of str and bytes objects are "salted" with an unpredictable random value. Although they remain constant within an individual Python process, they are not predictable between repeated invocations of Python.
>
> This is intended to provide protection against a denial-of-service caused by carefully chosen inputs that exploit the worst case performance of a dict insertion, $O(n^2)$ complexity. See http://ocert.org/advisories/ ocert-2011-003.html for details.
>
> Changing hash values affects the iteration order of sets. Python has never made guarantees about this ordering (and it typically varies between 32-bit and 64-bit builds).
>
> 另請參閱 PYTHONHASHSEED。

在 3.3 版的變更: Hash randomization is enabled by default.

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

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

3.3.2 Customizing attribute access

The following methods can be defined to customize the meaning of attribute access (use of, assignment to, or deletion of x.name) for class instances.

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

Called when the default attribute access fails with an AttributeError (either *__getattribute__()* raises

an `AttributeError` because *name* is not an instance attribute or an attribute in the class tree for `self`; or `__get__()` of a *name* property raises `AttributeError`). This method should either return the (computed) attribute value or raise an `AttributeError` exception. The `object` class itself does not provide this method.

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

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

Called unconditionally to implement attribute accesses for instances of the class. If the class also defines `__getattr__()`, the latter will not be called unless `__getattribute__()` either calls it explicitly or raises an `AttributeError`. This method should return the (computed) attribute value or raise an `AttributeError` exception. In order to avoid infinite recursion in this method, its implementation should always call the base class method with the same name to access any attributes it needs, for example, `object.__getattribute__(self, name)`.

> **ⓘ 備註**
>
> This method may still be bypassed when looking up special methods as the result of implicit invocation via language syntax or *built-in functions*. See *Special method lookup*.

For certain sensitive attribute accesses, raises an auditing event `object.__getattr__` with arguments `obj` and `name`.

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

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

If `__setattr__()` wants to assign to an instance attribute, it should call the base class method with the same name, for example, `object.__setattr__(self, name, value)`.

For certain sensitive attribute assignments, raises an auditing event `object.__setattr__` with arguments `obj`, `name`, `value`.

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

Like `__setattr__()` but for attribute deletion instead of assignment. This should only be implemented if `del obj.name` is meaningful for the object.

For certain sensitive attribute deletions, raises an auditing event `object.__delattr__` with arguments `obj` and `name`.

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

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

Customizing module attribute access

Special names `__getattr__` and `__dir__` can be also used to customize access to module attributes. The `__getattr__` function at the module level should accept one argument which is the name of an attribute and return the computed value or raise an `AttributeError`. If an attribute is not found on a module object through the normal lookup, i.e. `object.__getattribute__()`, then `__getattr__` is searched in the module `__dict__` before raising an `AttributeError`. If found, it is called with the attribute name and the result is returned.

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

For a more fine grained customization of the module behavior (setting attributes, properties, etc.), one can set the `__class__` attribute of a module object to a subclass of `types.ModuleType`. For example:

```python
import sys
from types import ModuleType

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

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

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

> ⓘ **備⿰**
>
> Defining module `__getattr__` and setting module `__class__` only affect lookups made using the attribute access syntax -- directly accessing the module globals (whether by code within the module, or via a reference to the module's globals dictionary) is unaffected.

在 3.5 版的變更: `__class__` 模組屬性現在是可寫入的。

在 3.7 版被加入: `__getattr__` 和 `__dir__` 模組屬性。

> ↪ **也參考**
>
> **PEP 562** - 模組 **__getattr__** 和 **__dir__**
>   Describes the `__getattr__` and `__dir__` functions on modules.

實作描述器

The following methods only apply when an instance of the class containing the method (a so-called *descriptor* class) appears in an *owner* class (the descriptor must be in either the owner's class dictionary or in the class dictionary for one of its parents). In the examples below, "the attribute" refers to the attribute whose name is the key of the property in the owner class' *__dict__*. The `object` class itself does not implement any of these protocols.

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

Called to get the attribute of the owner class (class attribute access) or of an instance of that class (instance attribute access). The optional *owner* argument is the owner class, while *instance* is the instance that the attribute was accessed through, or `None` when the attribute is accessed through the *owner*.

This method should return the computed attribute value or raise an `AttributeError` exception.

PEP 252 specifies that *__get__()* is callable with one or two arguments. Python's own built-in descriptors support this specification; however, it is likely that some third-party tools have descriptors that require both arguments. Python's own *__getattribute__()* implementation always passes in both arguments whether they are required or not.

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

Called to set the attribute on an instance *instance* of the owner class to a new value, *value*.

Note, adding *__set__()* or *__delete__()* changes the kind of descriptor to a "data descriptor". See *Invoking Descriptors* for more details.

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

> Called to delete the attribute on an instance *instance* of the owner class.

Instances of descriptors may also have the __objclass__ attribute present:

object.**__objclass__**

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

Invoking Descriptors

In general, a descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol: *__get__()*, *__set__()*, and *__delete__()*. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, a.x has a lookup chain starting with a.__dict__['x'], then type(a).__dict__['x'], and continuing through the base classes of type(a) excluding metaclasses.

However, if the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined and how they were called.

The starting point for descriptor invocation is a binding, a.x. How the arguments are assembled depends on a:

直接呼叫
> The simplest and least common call is when user code directly invokes a descriptor method: x.__get__(a).

Instance Binding
> If binding to an object instance, a.x is transformed into the call: type(a).__dict__['x'].__get__(a, type(a)).

Class Binding
> If binding to a class, A.x is transformed into the call: A.__dict__['x'].__get__(None, A).

Super Binding
> A dotted lookup such as super(A, a).x searches a.__class__.__mro__ for a base class B following A and then returns B.__dict__['x'].__get__(a, A). If not a descriptor, x is returned unchanged.

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

Python methods (including those decorated with @staticmethod and @classmethod) are implemented as non-data descriptors. Accordingly, instances can redefine and override methods. This allows individual instances to acquire behaviors that differ from other instances of the same class.

The property() function is implemented as a data descriptor. Accordingly, instances cannot override the behavior of a property.

__slots__

__slots__ allow us to explicitly declare data members (like properties) and deny the creation of *__dict__* and *__weakref__* (unless explicitly declared in *__slots__* or available in a parent.)

The space saved over using *__dict__* can be significant. Attribute lookup speed can be significantly improved as well.

object.**__slots__**

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

Notes on using *__slots__*:

- When inheriting from a class without *__slots__*, the *__dict__* and *__weakref__* attribute of the instances will always be accessible.

- Without a *__dict__* variable, instances cannot be assigned new variables not listed in the *__slots__* definition. Attempts to assign to an unlisted variable name raises AttributeError. If dynamic assignment of new variables is desired, then add '__dict__' to the sequence of strings in the *__slots__* declaration.

- Without a *__weakref__* variable for each instance, classes defining *__slots__* do not support weak references to its instances. If weak reference support is needed, then add '__weakref__' to the sequence of strings in the *__slots__* declaration.

- *__slots__* are implemented at the class level by creating *descriptors* for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by *__slots__*; otherwise, the class attribute would overwrite the descriptor assignment.

- The action of a *__slots__* declaration is not limited to the class where it is defined. *__slots__* declared in parents are available in child classes. However, instances of a child subclass will get a *__dict__* and *__weakref__* unless the subclass also defines *__slots__* (which should only contain names of any *additional* slots).

- If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the future, a check may be added to prevent this.

- TypeError will be raised if nonempty *__slots__* are defined for a class derived from a "variable-length" built-in type such as int, bytes, and tuple.

- Any non-string *iterable* may be assigned to *__slots__*.

- If a dictionary is used to assign *__slots__*, the dictionary keys will be used as the slot names. The values of the dictionary can be used to provide per-attribute docstrings that will be recognised by inspect.getdoc() and displayed in the output of help().

- *__class__* assignment works only if both classes have the same *__slots__*.

- Multiple inheritance with multiple slotted parent classes can be used, but only one parent is allowed to have attributes created by slots (the other bases must have empty slot layouts) - violations raise TypeError.

- If an *iterator* is used for *__slots__* then a *descriptor* is created for each of the iterator's values. However, the *__slots__* attribute will be an empty iterator.

3.3.3 Customizing class creation

Whenever a class inherits from another class, *__init_subclass__()* is called on the parent class. This way, it is possible to write classes which change the behavior of subclasses. This is closely related to class decorators, but where class decorators only affect the specific class they're applied to, __init_subclass__ solely applies to future subclasses of the class defining the method.

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

> This method is called whenever the containing class is subclassed. *cls* is then the new subclass. If defined as a normal instance method, this method is implicitly converted to a class method.

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

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

(繼續下一頁)

```
        cls.default_name = default_name

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

The default implementation `object.__init_subclass__` does nothing, but raises an error if it is called
with any arguments.

> **ⓘ 備註**
>
> The metaclass hint `metaclass` is consumed by the rest of the type machinery, and is never passed to
> `__init_subclass__` implementations. The actual metaclass (rather than the explicit hint) can be ac-
> cessed as `type(cls)`.

在 3.6 版被加入.

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

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

Automatically called at the time the owning class *owner* is created. The object has been assigned to *name* in
that class:

```
class A:
    x = C()   # 自動呼叫: x.__set_name__(A, 'x')
```

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

```
class A:
    pass

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

更多細節請見*Creating the class object*。

在 3.6 版被加入.

Metaclasses

By default, classes are constructed using `type()`. The class body is executed in a new namespace and the class name
is bound locally to the result of `type(name, bases, namespace)`.

The class creation process can be customized by passing the `metaclass` keyword argument in the class definition
line, or by inheriting from an existing class that included such an argument. In the following example, both `MyClass`
and `MySubclass` are instances of `Meta`:

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

Any other keyword arguments that are specified in the class definition are passed through to all metaclass operations
described below.

When a class definition is executed, the following steps occur:

- MRO entries are resolved;

- the appropriate metaclass is determined;

- the class namespace is prepared;

- the class body is executed;

- the class object is created.

Resolving MRO entries

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

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

> ↪ **也參考**
>
> **types.resolve_bases()**
>> Dynamically resolve bases that are not instances of `type`.
>
> **types.get_original_bases()**
>> Retrieve a class's "original bases" prior to modifications by *__mro_entries__()*.
>
> **PEP 560**
>> Core support for typing module and generic types.

Determining the appropriate metaclass

The appropriate metaclass for a class definition is determined as follows:

- if no bases and no explicit metaclass are given, then `type()` is used;

- if an explicit metaclass is given and it is *not* an instance of `type()`, then it is used directly as the metaclass;

- if an instance of `type()` is given as the explicit metaclass, or bases are defined, then the most derived metaclass is used.

The most derived metaclass is selected from the explicitly specified metaclass (if any) and the metaclasses (i.e. `type(cls)`) of all specified base classes. The most derived metaclass is one which is a subtype of *all* of these candidate metaclasses. If none of the candidate metaclasses meets that criterion, then the class definition will fail with `TypeError`.

Preparing the class namespace

Once the appropriate metaclass has been identified, then the class namespace is prepared. If the metaclass has a `__prepare__` attribute, it is called as `namespace = metaclass.__prepare__(name, bases, **kwds)` (where the additional keyword arguments, if any, come from the class definition). The `__prepare__` method should be implemented as a `classmethod`. The namespace returned by `__prepare__` is passed in to `__new__`, but when the final class object is created the namespace is copied into a new `dict`.

If the metaclass has no `__prepare__` attribute, then the class namespace is initialised as an empty ordered mapping.

> ↪ **也參考**
>
> **PEP 3115 - Metaclasses in Python 3000**
>> Introduced the `__prepare__` namespace hook

Executing the class body

The class body is executed (approximately) as `exec(body, globals(), namespace)`. The key difference from a normal call to `exec()` is that lexical scoping allows the class body (including any methods) to reference names from the current and outer scopes when the class definition occurs inside a function.

However, even when the class definition occurs inside the function, methods defined inside the class still cannot see names defined at the class scope. Class variables must be accessed through the first parameter of instance or class methods, or through the implicit lexically scoped `__class__` reference described in the next section.

Creating the class object

Once the class namespace has been populated by executing the class body, the class object is created by calling `metaclass(name, bases, namespace, **kwds)` (the additional keywords passed here are the same as those passed to `__prepare__`).

This class object is the one that will be referenced by the zero-argument form of `super()`. `__class__` is an implicit closure reference created by the compiler if any methods in a class body refer to either `__class__` or `super`. This allows the zero argument form of `super()` to correctly identify the class being defined based on lexical scoping, while the class or instance that was used to make the current call is identified based on the first argument passed to the method.

CPython 實作細節：In CPython 3.6 and later, the `__class__` cell is passed to the metaclass as a `__classcell__` entry in the class namespace. If present, this must be propagated up to the `type.__new__` call in order for the class to be initialised correctly. Failing to do so will result in a `RuntimeError` in Python 3.8.

When using the default metaclass `type`, or any metaclass that ultimately calls `type.__new__`, the following additional customization steps are invoked after creating the class object:

1) The `type.__new__` method collects all of the attributes in the class namespace that define a `__set_name__()` method;

2) Those `__set_name__` methods are called with the class being defined and the assigned name of that particular attribute;

3) The `__init_subclass__()` hook is called on the immediate parent of the new class in its method resolution order.

After the class object is created, it is passed to the class decorators included in the class definition (if any) and the resulting object is bound in the local namespace as the defined class.

When a new class is created by `type.__new__`, the object provided as the namespace parameter is copied to a new ordered mapping and the original object is discarded. The new copy is wrapped in a read-only proxy, which becomes the `__dict__` attribute of the class object.

> **➤ 也參考**
>
> **PEP 3135** - **New super**
> Describes the implicit `__class__` closure reference

Uses for metaclasses

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

3.3.4 Customizing instance and subclass checks

The following methods are used to override the default behavior of the `isinstance()` and `issubclass()` built-in functions.

In particular, the metaclass `abc.ABCMeta` implements these methods in order to allow the addition of Abstract Base Classes (ABCs) as "virtual base classes" to any class or type (including built-in types), including other ABCs.

---

type.**__instancecheck__**(*self*, *instance*)

> Return true if *instance* should be considered a (direct or indirect) instance of *class*. If defined, called to implement isinstance(instance, class).

type.**__subclasscheck__**(*self*, *subclass*)

> Return true if *subclass* should be considered a (direct or indirect) subclass of *class*. If defined, called to implement issubclass(subclass, class).

Note that these methods are looked up on the type (metaclass) of a class. They cannot be defined as class methods in the actual class. This is consistent with the lookup of special methods that are called on instances, only in this case the instance is itself a class.

> **↪ 也參考**
>
> **PEP 3119 - Introducing Abstract Base Classes**
> Includes the specification for customizing isinstance() and issubclass() behavior through *__instancecheck__()* and *__subclasscheck__()*, with motivation for this functionality in the context of adding Abstract Base Classes (see the abc module) to the language.

3.3.5 Emulating generic types

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

> **↪ 也參考**
>
> **PEP 484 - Type Hints**
> Introducing Python's framework for type annotations
>
> **泛型⬚名型⬚**
> Documentation for objects representing parameterized generic classes
>
> **Generics, user-defined generics and typing.Generic**
> Documentation on how to implement generic classes that can be parameterized at runtime and understood by static type-checkers.

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

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

> Return an object representing the specialization of a generic class by type arguments found in *key*.

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

The purpose of *__class_getitem__*

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

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

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

__*class_getitem*__ versus __*getitem*__

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

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

```python
from inspect import isclass

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

    class_of_obj = type(obj)

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

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

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

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

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

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

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

```
>>> # and the result is not a GenericAlias object:
>>> Menu['SPAM']
<Menu.SPAM: 'spam'>
>>> type(Menu['SPAM'])
<enum 'Menu'>
```

> ↪ **也參考**
>
> **PEP 560** - **Core Support for typing module and generic types**
>   Introducing __class_getitem__(), and outlining when a *subscription* results in __class_getitem__() being called instead of __getitem__()

3.3.6 Emulating callable objects

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

> Called when the instance is "called" as a function; if this method is defined, x(arg1, arg2, ...) roughly translates to type(x).__call__(x, arg1, ...). The object class itself does not provide this method.

3.3.7 Emulating container types

The following methods can be defined to implement container objects. None of them are provided by the object class itself. Containers usually are *sequences* (such as lists or tuples) or *mappings* (like *dictionaries*), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers k for which $0 <= k < N$ where N is the length of the sequence, or slice objects, which define a range of items. It is also recommended that mappings provide the methods keys(), values(), items(), get(), clear(), setdefault(), pop(), popitem(), copy(), and update() behaving similar to those for Python's standard dictionary objects. The collections.abc module provides a MutableMapping *abstract base class* to help create those methods from a base set of __getitem__(), __setitem__(), __delitem__(), and keys(). Mutable sequences should provide methods append(), count(), index(), extend(), insert(), pop(), remove(), reverse() and sort(), like Python standard list objects. Finally, sequence types should implement addition (meaning concatenation) and multiplication (meaning repetition) by defining the methods __add__(), __radd__(), __iadd__(), __mul__(), __rmul__() and __imul__() described below; they should not define other numerical operators. It is recommended that both mappings and sequences implement the __contains__() method to allow efficient use of the in operator; for mappings, in should search the mapping's keys; for sequences, it should search through the values. It is further recommended that both mappings and sequences implement the __iter__() method to allow efficient iteration through the container; for mappings, __iter__() should iterate through the object's keys; for sequences, it should iterate through the values.

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

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

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

object.**__length_hint__**(*self*)

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

> 在 3.4 版被加入.

> ℹ️ **備⬚**
>
> Slicing is done exclusively with the following three methods. A call like
>
> ```
> a[1:2] = b
> ```
>
> is translated to
>
> ```
> a[slice(1, 2, None)] = b
> ```
>
> and so forth. Missing slice items are always filled in with `None`.

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

> Called to implement evaluation of `self[key]`. For *sequence* types, the accepted keys should be integers. Optionally, they may support `slice` objects as well. Negative index support is also optional. If *key* is of an inappropriate type, `TypeError` may be raised; if *key* is a value outside the set of indexes for the sequence (after any special interpretation of negative values), `IndexError` should be raised. For *mapping* types, if *key* is missing (not in the container), `KeyError` should be raised.
>
> > ℹ️ **備⬚**
> >
> > `for` loops expect that an `IndexError` will be raised for illegal indexes to allow proper detection of the end of the sequence.
>
> > ℹ️ **備⬚**
> >
> > When *subscripting* a *class*, the special class method *__class_getitem__()* may be called instead of __getitem__(). See *__class_getitem__ versus __getitem__* for more details.

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

> Called to implement assignment to `self[key]`. Same note as for *__getitem__()*. This should only be implemented for mappings if the objects support changes to the values for keys, or if new keys can be added, or for sequences if elements can be replaced. The same exceptions should be raised for improper *key* values as for the *__getitem__()* method.

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

> Called to implement deletion of `self[key]`. Same note as for *__getitem__()*. This should only be implemented for mappings if the objects support removal of keys, or for sequences if elements can be removed from the sequence. The same exceptions should be raised for improper *key* values as for the *__getitem__()* method.

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

> Called by dict.*__getitem__()* to implement `self[key]` for dict subclasses when key is not in the dictionary.

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

> This method is called when an *iterator* is required for a container. This method should return a new iterator object that can iterate over all the objects in the container. For mappings, it should iterate over the keys of the container.

object.**__reversed__**(*self*)

> Called (if present) by the `reversed()` built-in to implement reverse iteration. It should return a new iterator object that iterates over all the objects in the container in reverse order.
>
> If the *__reversed__()* method is not provided, the `reversed()` built-in will fall back to using the sequence protocol (*__len__()* and *__getitem__()*). Objects that support the sequence protocol should only provide *__reversed__()* if they can provide an implementation that is more efficient than the one provided by `reversed()`.

The membership test operators (*in* and *not in*) are normally implemented as an iteration through a container. However, container objects can supply the following special method with a more efficient implementation, which also does not require the object be iterable.

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

> Called to implement membership test operators. Should return true if *item* is in *self*, false otherwise. For mapping objects, this should consider the keys of the mapping rather than the values or the key-item pairs.

> For objects that don't define *__contains__()*, the membership test first tries iteration via *__iter__()*, then the old sequence iteration protocol via *__getitem__()*, see *this section in the language reference*.

3.3.8 Emulating numeric types

The following methods can be defined to emulate numeric objects. Methods corresponding to operations that are not supported by the particular kind of number implemented (e.g., bitwise operations for non-integral numbers) should be left undefined.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

These methods are called to implement the binary arithmetic operations (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation[3] and the operands are of different types.[4] For instance, to evaluate the expression x - y, where *y* is an instance of a class that has an *__rsub__()* method, type(y).__rsub__(y, x) is called if type(x).__sub__(x, y) returns NotImplemented.

Note that ternary pow() will not try calling *__rpow__()* (the coercion rules would become too complicated).

> ℹ️ 備⿰
>
> If the right operand's type is a subclass of the left operand's type and that subclass provides a different implementation of the reflected method for the operation, this method will be called before the left operand's non-reflected method. This behavior allows subclasses to override their ancestors' operations.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Called to implement the unary arithmetic operations (-, +, abs() and ~).

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

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

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

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

---

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

[4] For operands of the same type, it is assumed that if the non-reflected method -- such as *__add__()* -- fails then the overall operation is not supported, which is why the reflected method is not called.

---

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

> Called to implement `operator.index()`, and whenever Python needs to losslessly convert the numeric object to an integer object (such as in slicing, or in the built-in `bin()`, `hex()` and `oct()` functions). Presence of this method indicates that the numeric object is an integer type. Must return an integer.
>
> If *__int__()*, *__float__()* and *__complex__()* are not defined then corresponding built-in functions `int()`, `float()` and `complex()` fall back to *__index__()*.

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

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

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

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

> Called to implement the built-in function `round()` and math functions `trunc()`, `floor()` and `ceil()`. Unless *ndigits* is passed to `__round__()` all these methods should return the value of the object truncated to an `Integral` (typically an `int`).
>
> The built-in function `int()` falls back to *__trunc__()* if neither *__int__()* nor *__index__()* is defined.
>
> 在 3.11 版的變更: The delegation of `int()` to *__trunc__()* is deprecated.

3.3.9 With 陳述式的情境管理器

A *context manager* is an object that defines the runtime context to be established when executing a *with* statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. Context managers are normally invoked using the `with` statement (described in section *with 陳述式*), but can also be used by directly invoking their methods.

Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.

For more information on context managers, see typecontextmanager. The `object` class itself does not provide the context manager methods.

object.**__enter__**(*self*)

> Enter the runtime context related to this object. The *with* statement will bind this method's return value to the target(s) specified in the `as` clause of the statement, if any.

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

> Exit the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be `None`.
>
> If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.
>
> Note that *__exit__()* methods should not reraise the passed-in exception; this is the caller's responsibility.

> ↪ **也參考**
>
> **PEP 343** - ”with” 陳述式
> > The specification, background, and examples for the Python *with* statement.

3.3.10 Customizing positional arguments in class pattern matching

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

object.**__match_args__**

> This class variable can be assigned a tuple of strings. When this class is used in a class pattern with positional arguments, each positional argument will be converted into a keyword argument, using the corresponding value in *__match_args__* as the keyword. The absence of this attribute is equivalent to setting it to `()`.

For example, if `MyClass.__match_args__` is `("left", "center", "right")` that means that `case MyClass(x, y)` is equivalent to `case MyClass(left=x, center=y)`. Note that the number of arguments in the pattern must be smaller than or equal to the number of elements in *__match_args__*; if it is larger, the pattern match attempt will raise a `TypeError`.

在 3.10 版被加入.

> ↪ **也參考**
>
> **PEP 634** - **Structural Pattern Matching**
> > The specification for the Python `match` statement.

3.3.11 Emulating buffer types

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

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

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

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

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

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

在 3.12 版被加入.

> ↪ **也參考**
>
> **PEP 688** - **Making the buffer protocol accessible in Python**
> > Introduces the Python __buffer__ and __release_buffer__ methods.
>
> **collections.abc.Buffer**
> > ABC for buffer types.

3.3.12 Special method lookup

For custom classes, implicit invocations of special methods are only guaranteed to work correctly if defined on an object's type, not in the object's instance dictionary. That behaviour is the reason why the following code raises an exception:

```
>>> class C:
...     pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
```

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

The rationale behind this behaviour lies with a number of special methods such as *__hash__()* and *__repr__()* that are implemented by all objects, including type objects. If the implicit lookup of these methods used the conventional lookup process, they would fail when invoked on the type object itself:

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

Incorrectly attempting to invoke an unbound method of a class in this way is sometimes referred to as 'metaclass confusion', and is avoided by bypassing the instance when looking up special methods:

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

In addition to bypassing any instance attributes in the interest of correctness, implicit special method lookup generally also bypasses the *__getattribute__()* method even of the object's metaclass:

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

Bypassing the *__getattribute__()* machinery in this fashion provides significant scope for speed optimisations within the interpreter, at the cost of some flexibility in the handling of special methods (the special method *must* be set on the class object itself in order to be consistently invoked by the interpreter).

3.4 協程

3.4.1 Awaitable Objects

An *awaitable* object generally implements an *__await__()* method. *Coroutine objects* returned from *async def* functions are awaitable.

> **ⓘ 備⽈**
>
> The *generator iterator* objects returned from generators decorated with `types.coroutine()` are also awaitable, but they do not implement `__await__()`.

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

> Must return an *iterator*. Should be used to implement *awaitable* objects. For instance, `asyncio.Future` implements this method to be compatible with the `await` expression. The `object` class itself is not awaitable and does not provide this method.
>
> > **ⓘ 備⽈**
> >
> > The language doesn't place any restriction on the type or value of the objects yielded by the iterator returned by `__await__`, as this is specific to the implementation of the asynchronous execution framework (e.g. `asyncio`) that will be managing the *awaitable* object.

在 3.5 版被加入.

> **↪ 也參考**
>
> **PEP 492** for additional information about awaitable objects.

3.4.2 協程物件

Coroutine objects are *awaitable* objects. A coroutine's execution can be controlled by calling `__await__()` and iterating over the result. When the coroutine has finished executing and returns, the iterator raises `StopIteration`, and the exception's `value` attribute holds the return value. If the coroutine raises an exception, it is propagated by the iterator. Coroutines should not directly raise unhandled `StopIteration` exceptions.

Coroutines also have the methods listed below, which are analogous to those of generators (see *Generator-iterator methods*). However, unlike generators, coroutines do not directly support iteration.

在 3.5.2 版的變更: It is a `RuntimeError` to await on a coroutine more than once.

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

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

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

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

> Raises the specified exception in the coroutine. This method delegates to the `throw()` method of the iterator that caused the coroutine to suspend, if it has such a method. Otherwise, the exception is raised at the suspension point. The result (return value, `StopIteration`, or other exception) is the same as when iterating over the `__await__()` return value, described above. If the exception is not caught in the coroutine, it propagates back to the caller.
>
> 在 3.12 版的變更: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

coroutine.**close**()

> Causes the coroutine to clean itself up and exit. If the coroutine is suspended, this method first delegates to the `close()` method of the iterator that caused the coroutine to suspend, if it has such a method. Then it raises `GeneratorExit` at the suspension point, causing the coroutine to immediately clean itself up. Finally, the coroutine is marked as having finished executing, even if it was never started.

Coroutine objects are automatically closed using the above process when they are about to be destroyed.

3.4.3 Asynchronous Iterators

An *asynchronous iterator* can call asynchronous code in its __anext__ method.

Asynchronous iterators can be used in an *async for* statement.

The object class itself does not provide these methods.

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

>   Must return an *asynchronous iterator* object.

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

>   Must return an *awaitable* resulting in a next value of the iterator. Should raise a StopAsyncIteration error when the iteration is over.

An example of an asynchronous iterable object:

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

    def __aiter__(self):
        return self

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

在 3.5 版被加入.

在 3.7 版的變更: Prior to Python 3.7, *__aiter__()* could return an *awaitable* that would resolve to an *asynchronous iterator*.

Starting with Python 3.7, *__aiter__()* must return an asynchronous iterator object. Returning anything else will result in a TypeError error.

3.4.4 Asynchronous Context Managers

An *asynchronous context manager* is a *context manager* that is able to suspend execution in its __aenter__ and __aexit__ methods.

Asynchronous context managers can be used in an *async with* statement.

The object class itself does not provide these methods.

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

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

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

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

An example of an asynchronous context manager class:

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

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

在 3.5 版被加入.

𝔽解

3.4. 協程

𝔽解

執行模型

4.1 Structure of a program

A Python program is constructed from code blocks. A *block* is a piece of Python program text that is executed as a unit. The following are blocks: a module, a function body, and a class definition. Each command typed interactively is a block. A script file (a file given as standard input to the interpreter or specified as a command line argument to the interpreter) is a code block. A script command (a command specified on the interpreter command line with the `-c` option) is a code block. A module run as a top level script (as module `__main__`) from the command line using a `-m` argument is also a code block. The string argument passed to the built-in functions `eval()` and `exec()` is a code block.

A code block is executed in an *execution frame*. A frame contains some administrative information (used for debugging) and determines where and how execution continues after the code block's execution has completed.

4.2 Naming and binding

4.2.1 Binding of names

Names refer to objects. Names are introduced by name binding operations.

The following constructs bind names:

- formal parameters to functions,
- class definitions,
- function definitions,
- assignment expressions,
- *targets* that are identifiers if occurring in an assignment:
    - `for` loop header,
    - after `as` in a `with` statement, `except` clause, `except *` clause, or in the as-pattern in structural pattern matching,
    - in a capture pattern in structural pattern matching
- `import` 陳述式。
- `type` 陳述式。

- *type parameter lists*.

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

A target occurring in a `del` statement is also considered bound for this purpose (though the actual semantics are to unbind the name).

Each assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

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

Each occurrence of a name in the program text refers to the *binding* of that name established by the following name resolution rules.

4.2.2 Resolution of names

A *scope* defines the visibility of a name within a block. If a local variable is defined in a block, its scope includes that block. If the definition occurs in a function block, the scope extends to any blocks contained within the defining one, unless a contained block introduces a different binding for the name.

When a name is used in a code block, it is resolved using the nearest enclosing scope. The set of all such scopes visible to a code block is called the block's *environment*.

When a name is not found at all, a `NameError` exception is raised. If the current scope is a function scope, and the name refers to a local variable that has not yet been bound to a value at the point where the name is used, an `UnboundLocalError` exception is raised. `UnboundLocalError` is a subclass of `NameError`.

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

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

The `global` statement has the same scope as a name binding operation in the same block. If the nearest enclosing scope for a free variable contains a global statement, the free variable is treated as a global.

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

The namespace for a module is automatically created the first time a module is imported. The main module for a script is always called `__main__`.

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

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

However, the following will succeed:

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

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

4.2.3 　釋作用域 (annotation scopes)

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

Annotation scopes are used in the following contexts:

- Type parameter lists for *generic type aliases*.

- Type parameter lists for *generic functions*. A generic function's annotations are executed within the annotation scope, but its defaults and decorators are not.

- Type parameter lists for *generic classes*. A generic class's base classes and keyword arguments are executed within the annotation scope, but its decorators are not.

- The bounds, constraints, and default values for type parameters (*lazily evaluated*).

- The value of type aliases (*lazily evaluated*).

Annotation scopes differ from function scopes in the following ways:

- Annotation scopes have access to their enclosing class namespace. If an annotation scope is immediately within a class scope, or within another annotation scope that is immediately within a class scope, the code in the annotation scope can use names defined in the class scope as if it were executed directly within the class body. This contrasts with regular functions defined within classes, which cannot access names defined in the class scope.

- Expressions in annotation scopes cannot contain `yield`, yield from, `await`, or `:=` expressions. (These expressions are allowed in other scopes contained within the annotation scope.)

- Names defined in annotation scopes cannot be rebound with `nonlocal` statements in inner scopes. This includes only type parameters, as no other syntactic elements that can appear within annotation scopes can introduce new names.

- While annotation scopes have an internal name, that name is not reflected in the *qualified name* of objects defined within the scope. Instead, the `__qualname__` of such objects is as if the object were defined in the enclosing scope.

在 3.12 版被加入: Annotation scopes were introduced in Python 3.12 as part of **PEP 695**.

在 3.13 版的變更: Annotation scopes are also used for type parameter defaults, as introduced by **PEP 696**.

4.2.4 Lazy evaluation

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

舉例來　:

```
>>> type Alias = 1/0
>>> Alias.__value__
Traceback (most recent call last):
  ...
ZeroDivisionError: division by zero
```

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

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

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

```
from typing import Literal

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

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

在 3.12 版被加入.

4.2.5 Builtins and restricted execution

CPython 實作細節： Users should not touch `__builtins__`; it is strictly an implementation detail. Users wanting to override values in the builtins namespace should *import* the `builtins` module and modify its attributes appropriately.

The builtins namespace associated with the execution of a code block is actually found by looking up the name `__builtins__` in its global namespace; this should be a dictionary or a module (in the latter case the module's dictionary is used). By default, when in the `__main__` module, `__builtins__` is the built-in module `builtins`; when in any other module, `__builtins__` is an alias for the dictionary of the `builtins` module itself.

4.2.6 Interaction with dynamic features

Name resolution of free variables occurs at runtime, not at compile time. This means that the following code will print 42:

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

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

4.3 例外

Exceptions are a means of breaking out of the normal flow of control of a code block in order to handle errors or other exceptional conditions. An exception is *raised* at the point where the error is detected; it may be *handled* by the surrounding code block or by any code block that directly or indirectly invoked the code block where the error occurred.

---

[1] This limitation occurs because the code that is executed by these operations is not available at the time the module is compiled.

The Python interpreter raises an exception when it detects a run-time error (such as division by zero). A Python program can also explicitly raise an exception with the *raise* statement. Exception handlers are specified with the *try* ... *except* statement. The *finally* clause of such a statement can be used to specify cleanup code which does not handle the exception, but is executed whether an exception occurred or not in the preceding code.

Python uses the "termination" model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top).

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

Exceptions are identified by class instances. The *except* clause is selected depending on the class of the instance: it must reference the class of the instance or a *non-virtual base class* thereof. The instance can be received by the handler and can carry additional information about the exceptional condition.

> ⓘ 備⿰
>
> Exception messages are not part of the Python API. Their contents may change from one version of Python to the next without warning and should not be relied on by code which will run under multiple versions of the interpreter.

See also the description of the *try* statement in section *try 陳述式* and *raise* statement in section *raise 陳述式*.

⿰解

模組引入系統

一個 *module* 中的 Python 程式碼透過 *importing* 的過程來存取另一個模組中的程式碼。*import* 陳述式是調用 (invoke) 引入機制最常見的方法，但這不是唯一的方法。函式如 importlib.import_module() 以及Ⓕ建函式 __import__() 也可以用來調用引入機制。

import 陳述式結合了兩個操作：首先搜尋指定的模組，然後將搜尋結果Ⓕ結到本地作用域中的一個名稱。import 陳述式的搜尋操作被定義Ⓕ一個對 __import__() 函式的呼叫，Ⓕ帶有相應的引數。__import__() 的回傳值用於執行 import 陳述式的名稱Ⓕ結操作。有關名稱Ⓕ結操作的詳細資訊，請參見 import 陳述式。

直接呼叫 __import__() 只會執行模組搜尋操作，以及在找到時執行模組的建立操作。雖然某些副作用可能會發生，例如引入父套件 (parent package)，以及更新各種快取（包括 sys.modules），但只有 *import* 陳述式會執行名稱Ⓕ結操作。

當執行 *import* 陳述式時，會呼叫Ⓕ建的 __import__() 函式。其他調用引入系統的機制（如 importlib.import_module()）可以選擇略過 __import__()，Ⓕ使用它們自己的解Ⓕ方案來實作引入語意。

當模組首次被引入時，Python 會搜尋該模組，若找到則會建立一個模組物件[1]，Ⓕ對其進行初始化。如果找不到指定的模組，則會引發 ModuleNotFoundError。當引入機制被調用時，Python 會實作各種策略來搜尋指定的模組。這些策略可以透過使用以下章節描述的各種 hook（ⒻⒻ）來修改和擴展。

在 3.3 版的變更: 引入系統已被更新，以完全實作 **PEP 302** 的第二階段。不再有隱式引入機制——完整的引入系統已透過 sys.meta_path 公開。此外，原生命名空間套件支援（請參Ⓕ **PEP 420**）也已被實作。

5.1 **importlib**

importlib 模組提供了豐富的 API 來與引入系統互動。例如，importlib.import_module() 提供了一個比Ⓕ建的 __import__() 更推薦且更簡單的 API 來調用引入機制。更多詳細資訊請參Ⓕ importlib 函式庫文件。

5.2 **套件**

Python 只有一種類型的模組物件，且所有模組，無論其是使用 Python、C 還是其他語言實作，都是這種類型。Ⓕ了幫助組織模組Ⓕ提供命名階層，Python 導入了套件的概念。

---

[1] 參Ⓕ types.ModuleType。

你可以將套件視為檔案系統中的目錄，模組則是目錄中的檔案，但不要過於字面地理解這個比喻，因為套件和模組不一定來自檔案系統。為了方便解釋，我們將使用這個目錄和檔案的比喻。就像檔案系統目錄一樣，套件是分層組織的，套件本身可以包含子套件以及一般模組。

請記住，所有的套件都是模組，但並非所有模組都是套件。換句話說，套件只是一種特殊的模組。具體來說，任何包含 __path__ 屬性的模組都被視為套件。

所有模組都有一個名稱。子套件的名稱與其父套件名稱之間用一個點來分隔，類似於 Python 的標準屬性存取語法。因此，你可能會有一個名為 email 的套件，該套件又有一個名為 email.mime 的子套件，並且該子套件中有一個名為 email.mime.text 的模組。

5.2.1 一般套件

Python 定義了兩種類型的套件，一般套件和命名空間套件。一般套件是 Python 3.2 及更早版本中存在的傳統套件。一般套件通常實作成一個包含 __init__.py 檔案的目錄。當引入一般套件時，該 __init__.py 檔案會被隱式執行，其定義的物件會綁結到該套件的命名空間中的名稱。__init__.py 檔案可以包含與任何其他模組相同的 Python 程式碼，並且 Python 會在引入時為該模組增加一些額外的屬性。

例如，以下檔案系統布置定義了一個頂層的 parent 套件，該套件包含三個子套件:

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

引入 parent.one 將隱式執行 parent/__init__.py 和 parent/one/__init__.py。隨後引入 parent.two 或 parent.three 將分別執行 parent/two/__init__.py 和 parent/three/__init__.py。

5.2.2 命名空間套件

命名空間套件是由不同的部分 組成的，每個部分都為父套件提供一個子套件。這些部分可以位於檔案系統上的不同位置。部分可能也存在於壓縮檔案中、網路上，或 Python 在引入時搜尋的任何其他地方。命名空間套件不一定直接對應於檔案系統中的對象;它們可能是沒有具體表示的虛擬模組。

命名空間套件的 __path__ 屬性不使用普通的串列。它們使用自定義的可迭代型別，當父套件的路徑（或頂層套件的 sys.path）發生變化時，會在下一次引入嘗試時自動執行新一輪的套件部分搜尋。

在命名空間套件中，不存在 parent/__init__.py 檔案。實際上，在引入搜尋過程中可能會找到多個 parent 目錄，每個目錄由不同的部分提供。因此，parent/one 可能與 parent/two 不會實際位於一起。在這種情況下，每當引入頂層 parent 套件或其子套件之一時，Python 會為頂層 parent 套件建立一個命名空間套件。

有關命名空間套件的規範，請參見 **PEP 420**。

5.3 搜尋

在開始搜尋之前，Python 需要被引入模組（或套件，但在本討論中，兩者的區別無關緊要）的完整限定名稱 (qualified name)。此名稱可能來自 import 陳述式的各種引數，或來自 importlib.import_module() 或 __import__() 函式的參數。

此名稱將在引入搜尋的各個階段中使用，並且它可能是指向子模組的點分隔路徑，例如 foo.bar.baz。在這種情況下，Python 會首先嘗試引入 foo，然後是 foo.bar，最後是 foo.bar.baz。如果任何中間引入失敗，則會引發 ModuleNotFoundError。

5.3.1 模組快取

在引入搜尋過程中首先檢查的地方是 `sys.modules`。此對映用作所有先前引入過的模組的快取，包括中間路徑。因此，如果 `foo.bar.baz` 之前已被引入，`sys.modules` 將包含 `foo`、`foo.bar` 和 `foo.bar.baz` 的條目。每個鍵的值都是相應的模組物件。

在引入過程中，會在 `sys.modules` 中查找模組名稱，如果存在，則相關的值即滿足此引入的模組，此引入過程即完成。然而，如果值是 `None`，則會引發 `ModuleNotFoundError`。如果模組名稱不存在，Python 會繼續搜尋該模組。

`sys.modules` 是可寫入的。移除一個鍵可能不會銷毀相關聯的模組（因為其他模組可能持有對它的參照），但會使指定的模組的快取條目失效，導致 Python 在下一次引入該模組時重新搜尋。也可以將鍵賦值為 `None`，這會強制下一次引入該模組時引發 `ModuleNotFoundError`。

但請注意，如果你保留了對模組物件的參照，並在 `sys.modules` 中使其快取條目失效，然後重新引入指定的模組，這兩個模組物件將不會相同。相比之下，`importlib.reload()` 會重用相同的模組物件，並透過重新執行模組的程式碼來簡單地重新初始化模組內容。

5.3.2 尋檢器 (Finder) 與載入器 (Loader)

如果在 `sys.modules` 中找不到指定的模組，則會調用 Python 的引入協定來尋找並載入該模組。這個協定由兩個概念性物件組成，*尋檢器* 和*載入器*。尋檢器的任務是使用其已知的策略來確定是否能找到命名模組。實作這兩個介面的物件稱為*引入器 (importer)* ——當它們發現可以載入所請求的模組時，會回傳它們自己。

Python 包含多個預設的尋檢器和引入器。第一個尋檢器知道如何定位內建模組，第二個尋檢器知道如何定位凍結模組。第三個預設尋檢器會在*import path* 中搜尋模組。*import path* 是一個位置的列表，這些位置可能是檔案系統路徑或壓縮檔案，也可以擴展以搜尋任何可定位的資源，例如由 URL 識別的資源。

引入機制是可擴展的，因此可以增加新的尋檢器來擴展模組搜尋的範圍和作用域。

尋檢器實際上不會載入模組。如果它們能找到指定的模組，它們會回傳一個模組規格，這是一個模組的引入相關資訊的封裝，引入機制會在載入模組時使用這些資訊。

以下各節將更詳細地描述尋檢器和載入器的協定，包括如何建立和註冊新的尋檢器和載入器來擴展引入機制。

在 3.4 版的變更: Python 在之前的版本中，尋檢器會直接回傳*載入器*，而現在它們回傳的是包含載入器的模組規格。載入器仍在引入過程中使用，但其責任減少了。

5.3.3 引入鉤子 (Import hooks)

引入機制的設計是可擴展的；其主要機制是引入鉤子。引入鉤子有兩種類型：元鉤子 *(meta hooks)* 和引入路徑鉤子。

元鉤子會在引入處理的開始階段被呼叫，除了查找 `sys.modules` 快取外，其他引入處理還未發生時就會呼叫。這允許元鉤子覆蓋 `sys.path` 的處理、凍結模組，甚至是內建模組。元鉤子透過將新的尋檢器物件添加到 `sys.meta_path` 中來註冊，具體描述請參閱以下段落。

引入路徑鉤子被視為 `sys.path`（或 `package.__path__`）處理過程的一部分來呼叫，當遇到與其相關聯的路徑項目時就會被觸發。引入路徑鉤子透過將新的可呼叫對象增加到 `sys.path_hooks` 中來註冊，具體描述請參閱以下段落。

5.3.4 元路徑

當在 `sys.modules` 中找不到命名模組時，Python 接下來會搜尋 `sys.meta_path`，其中包含一個元路徑尋檢器物件串列。這些尋檢器會依次被查詢，看它們是否知道如何處理命名模組。元路徑尋檢器必須實作一個名為 `find_spec()` 的方法，該方法接收三個引數：名稱、引入路徑和（可選的）目標模組。元路徑尋檢器可以使用任何策略來確定它是否能處理命名模組。

如果元路徑尋檢器知道如何處理命名模組，它會回傳一個規格物件。如果它無法處理命名模組，則回傳 `None`。如果 `sys.meta_path` 的處理到達串列的末尾仍未回傳規格，則會引發 `ModuleNotFoundError`。任何其他引發的例外將直接向上傳播，並中止引入過程。

元路徑尋檢器的 `find_spec()` 方法會以兩個或三個引數來呼叫。第一個是被引入模組的完全限定名稱，例如 `foo.bar.baz`。第二個引數是用於模組搜尋的路徑條目。對於頂層模組，第二個引數是 `None`，但對於子模組或子套件，第二個引數是父套件的 `__path__` 屬性的值。如果無法存取相應的 `__path__` 屬性，將引發 `ModuleNotFoundError`。第三個引數是一個現有的模組物件，該物件將成⊡後續載入的目標。引入系統只會在重新載入時傳入目標模組。

對於一個引入請求，元路徑可能會被遍歷多次。例如，假設參與的模組都⊡有被快取，則引入 `foo.bar.baz` 將首先執行頂層引入，對每個元路徑尋檢器（mpf）呼叫 `mpf.find_spec("foo", None, None)`。當 `foo` 被引入後，將再次藉由遍歷元路徑引入 `foo.bar`，⊡呼叫 `mpf.find_spec("foo.bar", foo.__path__, None)`。當 `foo.bar` 被引入後，最後一次遍歷會呼叫 `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)`。

一些元路徑尋檢器僅支援頂層引入。當第二個引數傳入 `None` 以外的值時，這些引入器將始終回傳 `None`。

Python 的預設 `sys.meta_path` 有三個元路徑尋檢器，一個知道如何引入⊡建模組，一個知道如何引入凍結模組，還有一個知道如何從 *import path* 引入模組（即 *path based finder*）。

在 3.4 版的變更: 元路徑尋檢器的 `find_spec()` 方法取代了 `find_module()`，後者現在已被⊡用。雖然它將繼續正常工作，但引入機制僅在尋檢器未實作 `find_spec()` 時才會嘗試使用它。

在 3.10 版的變更: 引入系統現在使用 `find_module()` 時將引發 `ImportWarning`。

在 3.12 版的變更: `find_module()` 已被移除。請改用 `find_spec()`。

5.4 載入

如果找到模組規格，引入機制會在載入模組時使用該規格（以及它包含的載入器）。以下是引入過程中載入部分的大致情⊡:

```python
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    # 這⊡假設載入器上也會定義 'exec_module'
    module = spec.loader.create_module(spec)
if module is None:
    module = ModuleType(spec.name)
# 與引入相關的模組屬性會在此處設定:
_init_module_attrs(spec, module)

if spec.loader is None:
    # 不支援
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    # 命名空間套件
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
else:
    sys.modules[spec.name] = module
    try:
        spec.loader.exec_module(module)
    except BaseException:
        try:
            del sys.modules[spec.name]
        except KeyError:
            pass
        raise
return sys.modules[spec.name]
```

請注意下列細節：

- 如果 `sys.modules` 中已存在具有給定名稱的模組物件，引入會已回傳該物件。

- 在載入器執行模組程式碼之前，模組將已存在於 `sys.modules` 中。這一點至關重要，因⊡模組程式碼可能會（直接或間接）引入自己；事先將其增加到 `sys.modules` 可以預防類似無限遞⊡以及

多次重覆載入等情形。

- 如果載入失敗，只有載入失敗的模組會從 sys.modules 中⊞除。任何已存在於 sys.modules 快取中的模組，以及任何在載入失敗前成功載入的模組，都必須保留在快取中。此情形與重新載入不同，在重新載入時，即使載入失敗的模組也會保留在 sys.modules 中。

- 模組建立後、在執行之前，引入機制會設置與引入相關的模組屬性（在上面的⊞程式碼範例中⊞"_init_module_attrs"），具體⊞容在*之後的段落*會總結。

- 模組執行是載入過程中的關鍵時刻，此時模組的命名空間會被新增名稱。執行過程完全交由載入器處理，由其⊞定如何新增以及新增什⊞。

- 在載入過程中建立⊞傳遞給 exec_module() 的模組，可能不會是引入結束時回傳的模組[2]。

在 3.4 版的變更: 引入系統已接管載入器的模板 (boilerplate) 責任。之前是由 importlib.abc.Loader. load_module() 方法執行的。

5.4.1 載入器

模組載入器提供了載入的關鍵功能: 模組執行。引入機制會以單一引數（即要執行的模組物件）呼叫 importlib.abc.Loader.exec_module() 方法。任何從 exec_module() 回傳的值都會被忽略。

載入器必須滿足以下要求:

- 如果模組是 Python 模組（而非⊞建模組或動態載入的擴充），載入器應在模組的全域命名空間 (module.__dict__) 中執行該模組的程式碼。

- 如果載入器無法執行該模組，應引發 ImportError。不過，在 exec_module() 中引發的任何其他例外也會被傳播。

在許多情⊞下，尋檢器和載入器可以是同一個物件；在這種情⊞下，find_spec() 方法只需回傳一個載入器設⊞ self 的規格即可。

模組載入器可以選擇透過實作 create_module() 方法，在載入過程中建立模組物件。該方法接受一個引數，即模組規格，⊞回傳在載入過程中要使用的新的模組物件。create_module() 不需要在模組物件上設定任何屬性。如果該方法回傳 None，引入機制將自行建立新的模組。

在 3.4 版被加入: 載入器的 create_module() 方法。

在 3.4 版的變更: load_module() 方法已被 exec_module() 取代，引入機制已承擔所有載入的模板責任。

⊞了與現有的載入器相容，引入機制會在載入器未實作 exec_module() 且存在 load_module() 方法時使用該方法。然而，load_module() 已被⊞用，載入器應改⊞實作 exec_module()。

load_module() 方法除了執行模組外，還必須實作上述全部的模板載入功能。所有相同的限制依然適用，⊞且還有一些額外的⊞明:

- 如果 sys.modules 中已存在具有給定名稱的模組物件，載入器必須使用該模組（否則 importlib. reload() 將無法正常運作）。如果命名模組不存在於 sys.modules 中，載入器必須建立一個新的模組物件⊞將其新增至 sys.modules。

- 在載入器執行模組程式碼之前，該模組必須已存在於 sys.modules 中，以防止無限遞⊞或多次載入。

- 如果載入失敗，載入器必須移除已經插入到 sys.modules 中的任何模組，但**只能**移除失敗的模組（們），且僅在載入器本身明確載入這些模組時才需移除。

在 3.5 版的變更: 當 exec_module() 已定義但未定義 create_module() 時，將引發 DeprecationWarning。

在 3.6 版的變更: 當 exec_module() 已定義但未定義 create_module() 時，將引發 ImportError。

在 3.10 版的變更: 使用 load_module() 將引發 ImportWarning。

---

[2] The importlib implementation avoids using the return value directly. Instead, it gets the module object by looking the module name up in sys.modules. The indirect effect of this is that an imported module may replace itself in sys.modules. This is implementation-specific behavior that is not guaranteed to work in other Python implementations.

5.4.2 子模組

當使用任何機制（例如 importlib APIs、import 或 import-from 陳述式，或內建的 __import__()）載入子模組時，會將子模組物件綁結到父模組的命名空間中。例如，如果套件 spam 有一個子模組 foo，則在引入 spam.foo 之後，spam 將擁有一個名為 foo 的屬性，該屬性綁結到子模組。我們假設你有以下的目錄結構：

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

並且 spam/__init__.py 中包含以下程式碼：

```python
from .foo import Foo
```

那麼執行以下程式碼會將 foo 和 Foo 的名稱綁結到 spam 模組中：

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

鑒於 Python 相似的名稱綁結規則，這可能看起來有些出人意料，但這實際上是引入系統的一個基本特性。不變的是如果你擁有 sys.modules['spam'] 和 sys.modules['spam.foo']（就像上述引入後那樣），那麼後者必須作為前者的 foo 屬性出現。

5.4.3 模組規格

引入機制在引入過程中使用有關每個模組的各種資訊，尤其是在載入之前。大多數資訊對所有模組來說都是通用的。模組規格的目的是以每個模組為基礎封裝這些與引入相關的資訊。

在引入過程中使用規格允許在引入系統的各個組件之間傳遞狀態，例如在建立模組規格的尋檢器和執行該規格的載入器之間傳遞。最重要的是，這允許引入機制執行載入的模板操作，而在沒有模組規格的情況下，這些操作則是載入器的責任。

模組的規格以 *module.__spec__* 的形式公開。適當地設定 __spec__ 同樣適用於*在直譯器啟動期間初始化的模組*。唯一的例外是 __main__，其中 __spec__ 會*在某些情況下被設定成 None*。

有關模組規格內容的詳細資訊，請參見 ModuleSpec。

在 3.4 版被加入.

5.4.4 __path__ attributes on modules

The *__path__* attribute should be a (possibly empty) *sequence* of strings enumerating the locations where the package's submodules will be found. By definition, if a module has a __path__ attribute, it is a *package*.

A package's *__path__* attribute is used during imports of its subpackages. Within the import machinery, it functions much the same as sys.path, i.e. providing a list of locations to search for modules during import. However, __path__ is typically much more constrained than sys.path.

The same rules used for sys.path also apply to a package's __path__. sys.path_hooks (described below) are consulted when traversing a package's __path__.

A package's __init__.py file may set or alter the package's *__path__* attribute, and this was typically the way namespace packages were implemented prior to **PEP 420**. With the adoption of **PEP 420**, namespace packages no longer need to supply __init__.py files containing only __path__ manipulation code; the import machinery automatically sets __path__ correctly for the namespace package.

5.4.5 Module reprs

By default, all modules have a usable repr, however depending on the attributes set above, and in the module's spec, you can more explicitly control the repr of module objects.

If the module has a spec (`__spec__`), the import machinery will try to generate a repr from it. If that fails or there is no spec, the import system will craft a default repr using whatever information is available on the module. It will try to use the `module.__name__`, `module.__file__`, and `module.__loader__` as input into the repr, with defaults for whatever information is missing.

Here are the exact rules used:

- If the module has a `__spec__` attribute, the information in the spec is used to generate the repr. The "name", "loader", "origin", and "has_location" attributes are consulted.

- If the module has a `__file__` attribute, this is used as part of the module's repr.

- If the module has no `__file__` but does have a `__loader__` that is not `None`, then the loader's repr is used as part of the module's repr.

- Otherwise, just use the module's `__name__` in the repr.

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

5.4.6 Cached bytecode invalidation

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

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

在 3.7 版的變更: Added hash-based `.pyc` files. Previously, Python only supported timestamp-based invalidation of bytecode caches.

5.5 The Path Based Finder

As mentioned previously, Python comes with several default meta path finders. One of these, called the *path based finder* (`PathFinder`), searches an *import path*, which contains a list of *path entries*. Each path entry names a location to search for modules.

The path based finder itself doesn't know how to import anything. Instead, it traverses the individual path entries, associating each of them with a path entry finder that knows how to handle that particular kind of path.

The default set of path entry finders implement all the semantics for finding modules on the file system, handling special file types such as Python source code (`.py` files), Python byte code (`.pyc` files) and shared libraries (e.g. `.so` files). When supported by the `zipimport` module in the standard library, the default path entry finders also handle loading all of these file types (other than shared libraries) from zipfiles.

Path entries need not be limited to file system locations. They can refer to URLs, database queries, or any other location that can be specified as a string.

The path based finder provides additional hooks and protocols so that you can extend and customize the types of searchable path entries. For example, if you wanted to support path entries as network URLs, you could write a hook that implements HTTP semantics to find modules on the web. This hook (a callable) would return a *path entry finder* supporting the protocol described below, which was then used to get a loader for the module from the web.

A word of warning: this section and the previous both use the term *finder*, distinguishing between them by using the terms *meta path finder* and *path entry finder*. These two types of finders are very similar, support similar protocols, and function in similar ways during the import process, but it's important to keep in mind that they are subtly different. In particular, meta path finders operate at the beginning of the import process, as keyed off the `sys.meta_path` traversal.

By contrast, path entry finders are in a sense an implementation detail of the path based finder, and in fact, if the path based finder were to be removed from `sys.meta_path`, none of the path entry finder semantics would be invoked.

5.5.1 Path entry finders

The *path based finder* is responsible for finding and loading Python modules and packages whose location is specified with a string *path entry*. Most path entries name locations in the file system, but they need not be limited to this.

As a meta path finder, the *path based finder* implements the `find_spec()` protocol previously described, however it exposes additional hooks that can be used to customize how modules are found and loaded from the *import path*.

Three variables are used by the *path based finder*, `sys.path`, `sys.path_hooks` and `sys.path_importer_cache`. The `__path__` attributes on package objects are also used. These provide additional ways that the import machinery can be customized.

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

The *path based finder* is a *meta path finder*, so the import machinery begins the *import path* search by calling the path based finder's `find_spec()` method as described previously. When the `path` argument to `find_spec()` is given, it will be a list of string paths to traverse - typically a package's `__path__` attribute for an import within that package. If the `path` argument is `None`, this indicates a top level import and `sys.path` is used.

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

If the path entry is not present in the cache, the path based finder iterates over every callable in `sys.path_hooks`. Each of the *path entry hooks* in this list is called with a single argument, the path entry to be searched. This callable may either return a *path entry finder* that can handle the path entry, or it may raise `ImportError`. An `ImportError` is used by the path based finder to signal that the hook cannot find a *path entry finder* for that *path entry*. The exception is ignored and *import path* iteration continues. The hook should expect either a string or bytes object; the encoding of bytes objects is up to the hook (e.g. it may be a file system encoding, UTF-8, or something else), and if the hook cannot decode the argument, it should raise `ImportError`.

If `sys.path_hooks` iteration ends with no *path entry finder* being returned, then the path based finder's `find_spec()` method will store `None` in `sys.path_importer_cache` (to indicate that there is no finder for this path entry) and return `None`, indicating that this *meta path finder* could not find the module.

If a *path entry finder is* returned by one of the *path entry hook* callables on `sys.path_hooks`, then the following protocol is used to ask the finder for a module spec, which is then used when loading the module.

The current working directory -- denoted by an empty string -- is handled slightly differently from other entries on `sys.path`. First, if the current working directory is found to not exist, no value is stored in `sys.path_importer_cache`. Second, the value for the current working directory is looked up fresh for each module lookup. Third, the path used for `sys.path_importer_cache` and returned by `importlib.machinery.PathFinder.find_spec()` will be the actual current working directory and not the empty string.

5.5.2 Path entry finder protocol

In order to support imports of modules and initialized packages and also to contribute portions to namespace packages, path entry finders must implement the `find_spec()` method.

`find_spec()` takes two arguments: the fully qualified name of the module being imported, and the (optional) target module. `find_spec()` returns a fully populated spec for the module. This spec will always have "loader" set (with one exception).

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

在 3.4 版的變更: `find_spec()` replaced `find_loader()` and `find_module()`, both of which are now deprecated, but will be used if `find_spec()` is not defined.

Older path entry finders may implement one of these two deprecated methods instead of `find_spec()`. The methods are still respected for the sake of backward compatibility. However, if `find_spec()` is implemented on the path entry finder, the legacy methods are ignored.

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

For backwards compatibility with other implementations of the import protocol, many path entry finders also support the same, traditional `find_module()` method that meta path finders support. However path entry finder `find_module()` methods are never called with a `path` argument (they are expected to record the appropriate path information from the initial call to the path hook).

The `find_module()` method on path entry finders is deprecated, as it does not allow the path entry finder to contribute portions to namespace packages. If both `find_loader()` and `find_module()` exist on a path entry finder, the import system will always call `find_loader()` in preference to `find_module()`.

在 3.10 版的變更: Calls to `find_module()` and `find_loader()` by the import system will raise `ImportWarning`.

在 3.12 版的變更: `find_module()` and `find_loader()` have been removed.

5.6 Replacing the standard import system

The most reliable mechanism for replacing the entire import system is to delete the default contents of `sys.meta_path`, replacing them entirely with a custom meta path hook.

If it is acceptable to only alter the behaviour of import statements without affecting other APIs that access the import system, then replacing the builtin `__import__()` function may be sufficient. This technique may also be employed at the module level to only alter the behaviour of import statements within that module.

To selectively prevent the import of some modules from a hook early on the meta path (rather than disabling the standard import system entirely), it is sufficient to raise `ModuleNotFoundError` directly from `find_spec()` instead of returning `None`. The latter indicates that the meta path search should continue, while raising an exception terminates it immediately.

5.7 Package Relative Imports

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

```
package/
    __init__.py
    subpackage1/
        __init__.py
        moduleX.py
        moduleY.py
```

```
    subpackage2/
        __init__.py
        moduleZ.py
    moduleA.py
```

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

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

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

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

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

5.8 Special considerations for __main__

The `__main__` module is a special case relative to Python's import system. As noted *elsewhere*, the `__main__` module is directly initialized at interpreter startup, much like `sys` and `builtins`. However, unlike those two, it doesn't strictly qualify as a built-in module. This is because the manner in which `__main__` is initialized depends on the flags and other options with which the interpreter is invoked.

5.8.1 __main__.__spec__

Depending on how `__main__` is initialized, `__main__.__spec__` gets set appropriately or to `None`.

When Python is started with the `-m` option, `__spec__` is set to the module spec of the corresponding module or package. `__spec__` is also populated when the `__main__` module is loaded as part of executing a directory, zipfile or other `sys.path` entry.

In the remaining cases `__main__.__spec__` is set to `None`, as the code used to populate the `__main__` does not correspond directly with an importable module:

- interactive prompt
- `-c` 選項
- running from stdin
- running directly from a source or bytecode file

Note that `__main__.__spec__` is always `None` in the last case, *even if* the file could technically be imported directly as a module instead. Use the `-m` switch if valid module metadata is desired in `__main__`.

Note also that even when `__main__` corresponds with an importable module and `__main__.__spec__` is set accordingly, they're still considered *distinct* modules. This is due to the fact that blocks guarded by `if __name__ == "__main__":` checks only execute when the module is used to populate the `__main__` namespace, and not during normal import.

5.9 References

The import machinery has evolved considerably since Python's early days. The original specification for packages is still available to read, although some details have changed since the writing of that document.

The original specification for `sys.meta_path` was **PEP 302**, with subsequent extension in **PEP 420**.

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

PEP 366 describes the addition of the `__package__` attribute for explicit relative imports in main modules.

PEP 328 introduced absolute and explicit relative imports and initially proposed `__name__` for semantics **PEP 366** would eventually specify for `__package__`.

PEP 338 defines executing modules as scripts.

PEP 451 adds the encapsulation of per-module import state in spec objects. It also off-loads most of the boilerplate responsibilities of loaders back onto the import machinery. These changes allow the deprecation of several APIs in the import system and also addition of new methods to finders and loaders.

F解

<div align="right">

運算式

</div>

This chapter explains the meaning of the elements of expressions in Python.

Syntax Notes: In this and the following chapters, extended BNF notation will be used to describe syntax, not lexical analysis. When (one alternative of) a syntax rule has the form

```
name ::= othername
```

and no semantics are given, the semantics of this form of `name` are the same as for `othername`.

6.1 Arithmetic conversions

When a description of an arithmetic operator below uses the phrase "the numeric arguments are converted to a common type", this means that the operator implementation for built-in types works as follows:

- If either argument is a complex number, the other is converted to complex;

- otherwise, if either argument is a floating-point number, the other is converted to floating point;

- otherwise, both must be integers and no conversion is necessary.

Some additional rules apply for certain operators (e.g., a string as a left argument to the '%' operator). Extensions must define their own conversion behavior.

6.2 Atoms

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

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

6.2.1 Identifiers (Names)

An identifier occurring as an atom is a name. See section *Identifiers and keywords* for lexical definition and section *Naming and binding* for documentation of naming and binding.

When the name is bound to an object, evaluation of the atom yields that object. When a name is not bound, an attempt to evaluate it raises a `NameError` exception.

Private name mangling

When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a *private name* of that class.

> ⤳ **也參考**
>
> The *class specifications*.

More precisely, private names are transformed to a longer form before code is generated for them. If the transformed name is longer than 255 characters, implementation-defined truncation may happen.

The transformation is independent of the syntactical context in which the identifier is used but only the following private identifiers are mangled:

- Any name used as the name of a variable that is assigned or read or any name of an attribute being accessed.

  The `__name__` attribute of nested functions, classes, and type aliases is however not mangled.

- The name of imported modules, e.g., `__spam` in `import __spam`. If the module is part of a package (i.e., its name contains a dot), the name is *not* mangled, e.g., the `__foo` in `import __foo.bar` is not mangled.

- The name of an imported member, e.g., `__f` in `from spam import __f`.

The transformation rule is defined as follows:

- The class name, with leading underscores removed and a single leading underscore inserted, is inserted in front of the identifier, e.g., the identifier `__spam` occurring in a class named `Foo`, `_Foo` or `__Foo` is transformed to `_Foo__spam`.

- If the class name consists only of underscores, the transformation is the identity, e.g., the identifier `__spam` occurring in a class named `_` or `__` is left as is.

6.2.2 Literals

Python supports string and bytes literals and various numeric literals:

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

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

All literals correspond to immutable data types, and hence the object's identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

6.2.3 Parenthesized forms

A parenthesized form is an optional expression list enclosed in parentheses:

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

A parenthesized expression list yields whatever that expression list yields: if the list contains at least one comma, it yields a tuple; otherwise, it yields the single expression that makes up the expression list.

An empty pair of parentheses yields an empty tuple object. Since tuples are immutable, the same rules as for literals apply (i.e., two occurrences of the empty tuple may or may not yield the same object).

Note that tuples are not formed by the parentheses, but rather by use of the comma. The exception is the empty tuple, for which parentheses *are* required --- allowing unparenthesized "nothing" in expressions would cause ambiguities and allow common typos to pass uncaught.

6.2.4 Displays for lists, sets and dictionaries

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

- either the container contents are listed explicitly, or

- they are computed via a set of looping and filtering instructions, called a *comprehension*.

Common syntax elements for comprehensions are:

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

The comprehension consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new container are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce an element each time the innermost block is reached.

However, aside from the iterable expression in the leftmost `for` clause, the comprehension is executed in a separate implicitly nested scope. This ensures that names assigned to in the target list don't "leak" into the enclosing scope.

The iterable expression in the leftmost `for` clause is evaluated directly in the enclosing scope and then passed as an argument to the implicitly nested scope. Subsequent `for` clauses and any filter condition in the leftmost `for` clause cannot be evaluated in the enclosing scope as they may depend on the values obtained from the leftmost iterable. For example: `[x*y for x in range(10) for y in range(x, x+10)]`.

To ensure the comprehension always results in a container of the appropriate type, `yield` and `yield from` expressions are prohibited in the implicitly nested scope.

Since Python 3.6, in an *async def* function, an `async for` clause may be used to iterate over a *asynchronous iterator*. A comprehension in an `async def` function may consist of either a `for` or `async for` clause following the leading expression, may contain additional `for` or `async for` clauses, and may also use *await* expressions.

If a comprehension contains `async for` clauses, or if it contains `await` expressions or other asynchronous comprehensions anywhere except the iterable expression in the leftmost `for` clause, it is called an *asynchronous comprehension*. An asynchronous comprehension may suspend the execution of the coroutine function in which it appears. See also **PEP 530**.

在 3.6 版被加入: Asynchronous comprehensions were introduced.

在 3.8 版的變更: `yield` and `yield from` prohibited in the implicitly nested scope.

在 3.11 版的變更: Asynchronous comprehensions are now allowed inside comprehensions in asynchronous functions. Outer comprehensions implicitly become asynchronous.

6.2.5 List displays

A list display is a possibly empty series of expressions enclosed in square brackets:

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

A list display yields a new list object, the contents being specified by either a list of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a comprehension is supplied, the list is constructed from the elements resulting from the comprehension.

6.2.6 Set displays

A set display is denoted by curly braces and distinguishable from dictionary displays by the lack of colons separating keys and values:

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

A set display yields a new mutable set object, the contents being specified by either a sequence of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and added to the set object. When a comprehension is supplied, the set is constructed from the elements resulting from the comprehension.

An empty set cannot be constructed with `{}`; this literal constructs an empty dictionary.

6.2.7 Dictionary displays

A dictionary display is a possibly empty series of dict items (key/value pairs) enclosed in curly braces:

```
dict_display       ::= "{" [dict_item_list | dict_comprehension] "}"
dict_item_list     ::= dict_item ("," dict_item)* [","]
dict_item          ::= expression ":" expression | "**" or_expr
dict_comprehension ::= expression ":" expression comp_for
```

A dictionary display yields a new dictionary object.

If a comma-separated sequence of dict items is given, they are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding value. This means that you can specify the same key multiple times in the dict item list, and the final dictionary's value for that key will be the last one given.

A double asterisk `**` denotes *dictionary unpacking*. Its operand must be a *mapping*. Each mapping item is added to the new dictionary. Later values replace values already set by earlier dict items and earlier dictionary unpackings.

在 3.5 版被加入: Unpacking into dictionary displays, originally proposed by **PEP 448**.

A dict comprehension, in contrast to list and set comprehensions, needs two expressions separated with a colon followed by the usual "for" and "if" clauses. When the comprehension is run, the resulting key and value elements are inserted in the new dictionary in the order they are produced.

Restrictions on the types of the key values are listed earlier in section 標準型⑥階層. (To summarize, the key type should be *hashable*, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last value (textually rightmost in the display) stored for a given key value prevails.

在 3.8 版的變更: Prior to Python 3.8, in dict comprehensions, the evaluation order of key and value was not well-defined. In CPython, the value was evaluated before the key. Starting with 3.8, the key is evaluated before the value, as proposed by **PEP 572**.

6.2.8 Generator expressions

A generator expression is a compact generator notation in parentheses:

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

A generator expression yields a new generator object. Its syntax is the same as for comprehensions, except that it is enclosed in parentheses instead of brackets or curly braces.

Variables used in the generator expression are evaluated lazily when the `__next__()` method is called for the generator object (in the same fashion as normal generators). However, the iterable expression in the leftmost `for` clause is immediately evaluated, so that an error produced by it will be emitted at the point where the generator expression is defined, rather than at the point where the first value is retrieved. Subsequent `for` clauses and any filter condition in the leftmost `for` clause cannot be evaluated in the enclosing scope as they may depend on the values obtained from the leftmost iterable. For example: `(x*y for x in range(10) for y in range(x, x+10))`.

The parentheses can be omitted on calls with only one argument. See section *Calls* for details.

To avoid interfering with the expected operation of the generator expression itself, `yield` and `yield from` expressions are prohibited in the implicitly defined generator.

If a generator expression contains either `async for` clauses or *await* expressions it is called an *asynchronous generator expression*. An asynchronous generator expression returns a new asynchronous generator object, which is an asynchronous iterator (see *Asynchronous Iterators*).

在 3.6 版被加入: Asynchronous generator expressions were introduced.

在 3.7 版的變更: Prior to Python 3.7, asynchronous generator expressions could only appear in *async def* coroutines. Starting with 3.7, any function can use asynchronous generator expressions.

在 3.8 版的變更: `yield` and `yield from` prohibited in the implicitly nested scope.

6.2.9 Yield expressions

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

The yield expression is used when defining a *generator* function or an *asynchronous generator* function and thus can only be used in the body of a function definition. Using a yield expression in a function's body causes that function to be a generator function, and using it in an *async def* function's body causes that coroutine function to be an asynchronous generator function. For example:

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


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

Due to their side effects on the containing scope, `yield` expressions are not permitted as part of the implicitly defined scopes used to implement comprehensions and generator expressions.

在 3.8 版的變更: Yield expressions prohibited in the implicitly nested scopes used to implement comprehensions and generator expressions.

Generator functions are described below, while asynchronous generator functions are described separately in section 非同步⽣器函式.

When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of the generator function. The execution starts when one of the generator's methods is called. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of *yield_list* to the generator's caller, or None if *yield_list* is omitted. By suspended, we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by calling one of the generator's methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If *__next__()* is used (typically via either a *for* or the `next()` builtin) then the result is None. Otherwise, if *send()* is used, then the result will be the value passed in to that method.

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

Yield expressions are allowed anywhere in a *try* construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator's *close()* method will be called, allowing any pending *finally* clauses to execute.

When `yield from <expr>` is used, the supplied expression must be an iterable. The values produced by iterating that iterable are passed directly to the caller of the current generator's methods. Any values passed in with *send()* and any exceptions passed in with *throw()* are passed to the underlying iterator if it has the appropriate methods. If this is not the case, then *send()* will raise AttributeError or TypeError, while *throw()* will just raise the passed in exception immediately.

When the underlying iterator is complete, the `value` attribute of the raised StopIteration instance becomes the value of the yield expression. It can be either set explicitly when raising StopIteration, or automatically when the subiterator is a generator (by returning a value from the subgenerator).

在 3.3 版的變更: Added `yield from <expr>` to delegate control flow to a subiterator.

The parentheses may be omitted when the yield expression is the sole expression on the right hand side of an assignment statement.

> → **也參考**
>
> **PEP 255** - **Simple Generators**
> The proposal for adding generators and the `yield` statement to Python.
>
> **PEP 342** - **Coroutines via Enhanced Generators**
> The proposal to enhance the API and syntax of generators, making them usable as simple coroutines.
>
> **PEP 380** - **Syntax for Delegating to a Subgenerator**
> The proposal to introduce the `yield_from` syntax, making delegation to subgenerators easy.
>
> **PEP 525** - **Asynchronous Generators**
> The proposal that expanded on **PEP 492** by adding generator capabilities to coroutine functions.

Generator-iterator methods

This subsection describes the methods of a generator iterator. They can be used to control the execution of a generator function.

Note that calling any of the generator methods below when the generator is already executing raises a `ValueError` exception.

generator.**__next__**()

> Starts the execution of a generator function or resumes it at the last executed yield expression. When a generator function is resumed with a `__next__()` method, the current yield expression always evaluates to `None`. The execution then continues to the next yield expression, where the generator is suspended again, and the value of the `yield_list` is returned to `__next__()`'s caller. If the generator exits without yielding another value, a `StopIteration` exception is raised.
>
> This method is normally called implicitly, e.g. by a `for` loop, or by the built-in `next()` function.

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

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

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

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

> Raises an exception at the point where the generator was paused, and returns the next value yielded by the generator function. If the generator exits without yielding another value, a `StopIteration` exception is raised. If the generator function does not catch the passed-in exception, or raises a different exception, then that exception propagates to the caller.
>
> In typical use, this is called with a single exception instance similar to the way the `raise` keyword is used.
>
> For backwards compatibility, however, the second signature is supported, following a convention from older versions of Python. The *type* argument should be an exception class, and *value* should be an exception instance. If the *value* is not provided, the *type* constructor is called to get an instance. If *traceback* is provided, it is set on the exception, otherwise any existing `__traceback__` attribute stored in *value* may be cleared.
>
> 在 3.12 版的變更: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

generator.**close**()

> Raises a `GeneratorExit` at the point where the generator function was paused. If the generator function catches the exception and returns a value, this value is returned from `close()`. If the generator function

is already closed, or raises `GeneratorExit` (by not catching the exception), *close()* returns `None`. If the generator yields a value, a `RuntimeError` is raised. If the generator raises any other exception, it is propagated to the caller. If the generator has already exited due to an exception or normal exit, *close()* returns `None` and has no other effect.

在 3.13 版的變更: If a generator returns a value upon being closed, the value is returned by *close()*.

模組

Here is a simple example that demonstrates the behavior of generators and generator functions:

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

For examples using `yield from`, see pep-380 in "What's New in Python."

非同步生器函式

The presence of a yield expression in a function or method defined using *async def* further defines the function as an *asynchronous generator* function.

When an asynchronous generator function is called, it returns an asynchronous iterator known as an asynchronous generator object. That object then controls the execution of the generator function. An asynchronous generator object is typically used in an *async for* statement in a coroutine function analogously to how a generator object would be used in a *for* statement.

Calling one of the asynchronous generator's methods returns an *awaitable* object, and the execution starts when this object is awaited on. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of *yield_list* to the awaiting coroutine. As with a generator, suspension means that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by awaiting on the next object returned by the asynchronous generator's methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If *__anext__()* is used then the result is `None`. Otherwise, if *asend()* is used, then the result will be the value passed in to that method.

If an asynchronous generator happens to exit early by *break*, the caller task being cancelled, or other exceptions, the generator's async cleanup code will run and possibly raise exceptions or access context variables in an unexpected context--perhaps after the lifetime of tasks it depends, or during the event loop shutdown when the async-generator garbage collection hook is called. To prevent this, the caller must explicitly close the async generator by calling *aclose()* method to finalize the generator and ultimately detach it from the event loop.

In an asynchronous generator function, yield expressions are allowed anywhere in a `try` construct. However, if an asynchronous generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), then a yield expression within a `try` construct could result in a failure to execute pending `finally` clauses. In this case, it is the responsibility of the event loop or scheduler running the asynchronous generator to call the asynchronous generator-iterator's `aclose()` method and run the resulting coroutine object, thus allowing any pending `finally` clauses to execute.

To take care of finalization upon event loop termination, an event loop should define a *finalizer* function which takes an asynchronous generator-iterator and presumably calls `aclose()` and executes the coroutine. This *finalizer* may be registered by calling `sys.set_asyncgen_hooks()`. When first iterated over, an asynchronous generator-iterator will store the registered *finalizer* to be called upon finalization. For a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in Lib/asyncio/base_events.py.

The expression `yield from <expr>` is a syntax error when used in an asynchronous generator function.

Asynchronous generator-iterator methods

This subsection describes the methods of an asynchronous generator iterator, which are used to control the execution of a generator function.

async agen.**__anext__**()

> Returns an awaitable which when run starts to execute the asynchronous generator or resumes it at the last executed yield expression. When an asynchronous generator function is resumed with an `__anext__()` method, the current yield expression always evaluates to `None` in the returned awaitable, which when run will continue to the next yield expression. The value of the `yield_list` of the yield expression is the value of the `StopIteration` exception raised by the completing coroutine. If the asynchronous generator exits without yielding another value, the awaitable instead raises a `StopAsyncIteration` exception, signalling that the asynchronous iteration has completed.
>
> This method is normally called implicitly by a `async for` loop.

async agen.**asend**(*value*)

> Returns an awaitable which when run resumes the execution of the asynchronous generator. As with the `send()` method for a generator, this "sends" a value into the asynchronous generator function, and the *value* argument becomes the result of the current yield expression. The awaitable returned by the `asend()` method will return the next value yielded by the generator as the value of the raised `StopIteration`, or raises `StopAsyncIteration` if the asynchronous generator exits without yielding another value. When `asend()` is called to start the asynchronous generator, it must be called with `None` as the argument, because there is no yield expression that could receive the value.

async agen.**athrow**(*value*)

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

> Returns an awaitable that raises an exception of type `type` at the point where the asynchronous generator was paused, and returns the next value yielded by the generator function as the value of the raised `StopIteration` exception. If the asynchronous generator exits without yielding another value, a `StopAsyncIteration` exception is raised by the awaitable. If the generator function does not catch the passed-in exception, or raises a different exception, then when the awaitable is run that exception propagates to the caller of the awaitable.

> 在 3.12 版的變更: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

async agen.**aclose**()

> Returns an awaitable that when run will throw a `GeneratorExit` into the asynchronous generator function at the point where it was paused. If the asynchronous generator function then exits gracefully, is already closed, or raises `GeneratorExit` (by not catching the exception), then the returned awaitable will raise a `StopIteration` exception. Any further awaitables returned by subsequent calls to the asynchronous generator will raise a `StopAsyncIteration` exception. If the asynchronous generator yields a value, a `RuntimeError` is raised by the awaitable. If the asynchronous generator raises any other exception, it is propagated to the caller of the awaitable. If the asynchronous generator has already exited due to an exception or normal exit, then further calls to `aclose()` will return an awaitable that does nothing.

6.3 Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

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

6.3.1 Attribute references

An attribute reference is a primary followed by a period and a name:

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

The primary must evaluate to an object of a type that supports attribute references, which most objects do. This object is then asked to produce the attribute whose name is the identifier. The type and value produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

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

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

6.3.2 Subscriptions

The subscription of an instance of a *container class* will generally select an element from the container. The subscription of a *generic class* will generally return a GenericAlias object.

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

When an object is subscripted, the interpreter will evaluate the primary and the expression list.

The primary must evaluate to an object that supports subscription. An object may support subscription through defining one or both of `__getitem__()` and `__class_getitem__()`. When the primary is subscripted, the evaluated result of the expression list will be passed to one of these methods. For more details on when `__class_getitem__` is called instead of `__getitem__`, see *__class_getitem__ versus __getitem__*.

If the expression list contains at least one comma, or if any of the expressions are starred, the expression list will evaluate to a `tuple` containing the items of the expression list. Otherwise, the expression list will evaluate to the value of the list's sole member.

在 3.11 版的變更: Expressions in an expression list may be starred. See **PEP 646**.

For built-in objects, there are two types of objects that support subscription via `__getitem__()`:

1. Mappings. If the primary is a *mapping*, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key. An example of a builtin mapping class is the `dict` class.

2. Sequences. If the primary is a *sequence*, the expression list must evaluate to an `int` or a `slice` (as discussed in the following section). Examples of builtin sequence classes include the `str`, `list` and `tuple` classes.

The formal syntax makes no special provision for negative indices in *sequences*. However, built-in sequences all provide a `__getitem__()` method that interprets negative indices by adding the length of the sequence to the index so that, for example, `x[-1]` selects the last item of `x`. The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero). Since the support for negative indices and slicing occurs in the object's `__getitem__()` method, subclasses overriding this method will need to explicitly add that support.

A `string` is a special kind of sequence whose items are *characters*. A character is not a separate data type but a string of exactly one character.

6.3.3 Slicings

A slicing selects a range of items in a sequence object (e.g., a string, tuple or list). Slicings may be used as expressions or as targets in assignment or *del* statements. The syntax for a slicing:

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

There is ambiguity in the formal syntax here: anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation as a slicing (this is the case if the slice list contains no proper slice).

The semantics for a slicing are as follows. The primary is indexed (using the same *__getitem__()* method as normal subscription) with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of a proper slice is a slice object (see section 標準型階層) whose `start`, `stop` and `step` attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting `None` for missing expressions.

6.3.4 Calls

A call calls a callable object (e.g., a *function*) with a possibly empty series of *arguments*:

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

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

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and all objects having a *__call__()* method are callable). All argument expressions are evaluated before the call is attempted. Please refer to section 函式定義 for the syntax of formal *parameter* lists.

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a `TypeError` exception is raised. Otherwise, the argument is placed in the slot, filling it (even if the expression is `None`, it fills the slot). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that don't specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a `TypeError` exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

CPython 實作細節: An implementation may provide built-in functions whose positional parameters do not have names, even if they are 'named' for the purpose of documentation, and which therefore cannot be supplied by keyword. In CPython, this is the case for functions implemented in C that use `PyArg_ParseTuple()` to parse their arguments.

If there are more positional arguments than there are formal parameter slots, a `TypeError` exception is raised, unless a formal parameter using the syntax `*identifier` is present; in this case, that formal parameter receives a tuple containing the excess positional arguments (or an empty tuple if there were no excess positional arguments).

If any keyword argument does not correspond to a formal parameter name, a `TypeError` exception is raised, unless a formal parameter using the syntax `**identifier` is present; in this case, that formal parameter receives a dictionary containing the excess keyword arguments (using the keywords as keys and the argument values as corresponding values), or a (new) empty dictionary if there were no excess keyword arguments.

If the syntax `*expression` appears in the function call, `expression` must evaluate to an *iterable*. Elements from these iterables are treated as if they were additional positional arguments. For the call `f(x1, x2, *y, x3, x4)`, if *y* evaluates to a sequence *y1, ..., yM*, this is equivalent to a call with M+4 positional arguments *x1, x2, y1, ..., yM, x3, x4*.

A consequence of this is that although the `*expression` syntax may appear *after* explicit keyword arguments, it is processed *before* the keyword arguments (and any `**expression` arguments -- see below). So:

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

It is unusual for both keyword arguments and the `*expression` syntax to be used in the same call, so in practice this confusion does not often arise.

If the syntax `**expression` appears in the function call, `expression` must evaluate to a *mapping*, the contents of which are treated as additional keyword arguments. If a parameter matching a key has already been given a value (by an explicit keyword argument, or from another unpacking), a `TypeError` exception is raised.

When `**expression` is used, each key in this mapping must be a string. Each value from the mapping is assigned to the first formal parameter eligible for keyword assignment whose name is equal to the key. A key need not be a Python identifier (e.g. `"max-temp °F"` is acceptable, although it will not match any formal parameter that could be declared). If there is no match to a formal parameter the key-value pair is collected by the `**` parameter, if there is one, or if there is not, a `TypeError` exception is raised.

Formal parameters using the syntax `*identifier` or `**identifier` cannot be used as positional argument slots or as keyword argument names.

在 3.5 版的變更: Function calls accept any number of `*` and `**` unpackings, positional arguments may follow iterable unpackings (`*`), and keyword arguments may follow dictionary unpackings (`**`). Originally proposed by **PEP 448**.

A call always returns some value, possibly `None`, unless it raises an exception. How this value is computed depends on the type of the callable object.

If it is---

a user-defined function:
> The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section *函式定義*. When the code block executes a `return` statement, this specifies the return value of the function call. If execution reaches the end of the code block without executing a `return` statement, the return value is `None`.

a built-in function or method:
> The result is up to the interpreter; see built-in-funcs for the descriptions of built-in functions and methods.

a class object:
> A new instance of that class is returned.

a class instance method:
> The corresponding user-defined function is called, with an argument list that is one longer than the argument list of the call: the instance becomes the first argument.

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

6.4 Await expression

Suspend the execution of *coroutine* on an *awaitable* object. Can only be used inside a *coroutine function*.

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

在 3.5 版被加入.

6.5 The power operator

The power operator binds more tightly than unary operators on its left; it binds less tightly than unary operators on its right. The syntax is:

```
power ::= (await_expr | primary) ["**" u_expr]
```

Thus, in an unparenthesized sequence of power and unary operators, the operators are evaluated from right to left (this does not constrain the evaluation order for the operands): `-1**2` results in `-1`.

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

For int operands, the result has the same type as the operands unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, `10**2` returns `100`, but `10**-2` returns `0.01`.

Raising `0.0` to a negative power results in a `ZeroDivisionError`. Raising a negative number to a fractional power results in a `complex` number. (In earlier versions it raised a `ValueError`.)

This operation can be customized using the special `__pow__()` and `__rpow__()` methods.

6.6 Unary arithmetic and bitwise operations

All unary arithmetic and bitwise operations have the same priority:

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

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

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

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

In all three cases, if the argument does not have the proper type, a `TypeError` exception is raised.

6.7 Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. Apart from the power operator, there are only two levels, one for multiplicative operators and one for additive operators:

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

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

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

The `@` (at) operator is intended to be used for matrix multiplication. No builtin Python types implement this operator.

This operation can be customized using the special *__matmul__()* and *__rmatmul__()* methods.

在 3.5 版被加入.

The `/` (division) and `//` (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Division of integers yields a float, while floor division of integers results in an integer; the result is that of mathematical division with the 'floor' function applied to the result. Division by zero raises the `ZeroDivisionError` exception.

The division operation can be customized using the special *__truediv__()* and *__rtruediv__()* methods. The floor division operation can be customized using the special *__floordiv__()* and *__rfloordiv__()* methods.

The `%` (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating-point numbers, e.g., `3.14%0.7` equals `0.34` (since `3.14` equals `4*0.7 + 0.34`.) The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand[1].

The floor division and modulo operators are connected by the following identity: `x == (x//y)*y + (x%y)`. Floor division and modulo are also connected with the built-in function `divmod()`: `divmod(x, y) == (x//y, x%y)`.[2]

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

The *modulo* operation can be customized using the special *__mod__()* and *__rmod__()* methods.

The floor division operator, the modulo operator, and the `divmod()` function are not defined for complex numbers. Instead, convert to a floating-point number using the `abs()` function if appropriate.

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

This operation can be customized using the special *__add__()* and *__radd__()* methods.

The `-` (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common type.

This operation can be customized using the special *__sub__()* and *__rsub__()* methods.

---

[1] While `abs(x%y) < abs(y)` is true mathematically, for floats it may not be true numerically due to roundoff. For example, and assuming a platform on which a Python float is an IEEE 754 double-precision number, in order that `-1e-100 % 1e100` have the same sign as `1e100`, the computed result is `-1e-100 + 1e100`, which is numerically exactly equal to `1e100`. The function `math.fmod()` returns a result whose sign matches the sign of the first argument instead, and so returns `-1e-100` in this case. Which approach is more appropriate depends on the application.

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

6.8 Shifting operations

The shifting operations have lower priority than the arithmetic operations:

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

These operators accept integers as arguments. They shift the first argument to the left or right by the number of bits given by the second argument.

The left shift operation can be customized using the special `__lshift__()` and `__rlshift__()` methods. The right shift operation can be customized using the special `__rshift__()` and `__rrshift__()` methods.

A right shift by *n* bits is defined as floor division by `pow(2,n)`. A left shift by *n* bits is defined as multiplication with `pow(2,n)`.

6.9 Binary bitwise operations

Each of the three bitwise operations has a different priority level:

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

The `&` operator yields the bitwise AND of its arguments, which must be integers or one of them must be a custom object overriding `__and__()` or `__rand__()` special methods.

The `^` operator yields the bitwise XOR (exclusive OR) of its arguments, which must be integers or one of them must be a custom object overriding `__xor__()` or `__rxor__()` special methods.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be integers or one of them must be a custom object overriding `__or__()` or `__ror__()` special methods.

6.10 Comparisons

Unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also unlike C, expressions like `a < b < c` have the interpretation that is conventional in mathematics:

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

Comparisons yield boolean values: `True` or `False`. Custom *rich comparison methods* may return non-boolean values. In this case Python will call `bool()` on such value in boolean contexts.

Comparisons can be chained arbitrarily, e.g., `x < y <= z` is equivalent to `x < y and y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

Formally, if *a*, *b*, *c*, ..., *y*, *z* are expressions and *op1*, *op2*, ..., *opN* are comparison operators, then `a op1 b op2 c ... y opN z` is equivalent to `a op1 b and b op2 c and ... y opN z`, except that each expression is evaluated at most once.

Note that `a op1 b op2 c` doesn't imply any kind of comparison between *a* and *c*, so that, e.g., `x < y > z` is perfectly legal (though perhaps not pretty).

6.10.1 Value comparisons

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

Chapter 物件、數值和型別 states that objects have a value (in addition to type and identity). The value of an object is a rather abstract notion in Python: For example, there is no canonical access method for an object's value. Also,

there is no requirement that the value of an object should be constructed in a particular way, e.g. comprised of all its data attributes. Comparison operators implement a particular notion of what the value of an object is. One can think of them as defining the value of an object indirectly, by means of their comparison implementation.

Because all types are (direct or indirect) subtypes of `object`, they inherit the default comparison behavior from `object`. Types can customize their comparison behavior by implementing *rich comparison methods* like `__lt__()`, described in *Basic customization*.

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

A default order comparison (`<`, `>`, `<=`, and `>=`) is not provided; an attempt raises `TypeError`. A motivation for this default behavior is the lack of a similar invariant as for equality.

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

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

- Numbers of built-in numeric types (typesnumeric) and of the standard library types `fractions.Fraction` and `decimal.Decimal` can be compared within and across their types, with the restriction that complex numbers do not support order comparison. Within the limits of the types involved, they compare mathematically (algorithmically) correct without loss of precision.

  The not-a-number values `float('NaN')` and `decimal.Decimal('NaN')` are special. Any ordered comparison of a number to a not-a-number value is false. A counter-intuitive implication is that not-a-number values are not equal to themselves. For example, if `x = float('NaN')`, `3 < x`, `x < 3` and `x == x` are all false, while `x != x` is true. This behavior is compliant with IEEE 754.

- `None` and `NotImplemented` are singletons. **PEP 8** advises that comparisons for singletons should always be done with `is` or `is not`, never the equality operators.

- Binary sequences (instances of `bytes` or `bytearray`) can be compared within and across their types. They compare lexicographically using the numeric values of their elements.

- Strings (instances of `str`) compare lexicographically using the numerical Unicode code points (the result of the built-in function `ord()`) of their characters.[3]

  Strings and binary sequences cannot be directly compared.

- Sequences (instances of `tuple`, `list`, or `range`) can be compared only within each of their types, with the restriction that ranges do not support order comparison. Equality comparison across these types results in inequality, and ordering comparison across these types raises `TypeError`.

  Sequences compare lexicographically using comparison of corresponding elements. The built-in containers typically assume identical objects are equal to themselves. That lets them bypass equality tests for identical objects to improve performance and to maintain their internal invariants.

  Lexicographical comparison between built-in collections works as follows:

  - For two collections to compare equal, they must be of the same type, have the same length, and each pair of corresponding elements must compare equal (for example, `[1,2] == (1,2)` is false because the type is not the same).

---

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

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

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

    – Collections that support order comparison are ordered the same as their first unequal elements (for example, `[1,2,x] <= [1,2,y]` has the same value as `x <= y`). If a corresponding element does not exist, the shorter collection is ordered first (for example, `[1,2] < [1,2,3]` is true).

- Mappings (instances of `dict`) compare equal if and only if they have equal `(key, value)` pairs. Equality comparison of the keys and values enforces reflexivity.

  Order comparisons (<, >, <=, and >=) raise `TypeError`.

- Sets (instances of `set` or `frozenset`) can be compared within and across their types.

  They define order comparison operators to mean subset and superset tests. Those relations do not define total orderings (for example, the two sets `{1,2}` and `{2,3}` are not equal, nor subsets of one another, nor supersets of one another). Accordingly, sets are not appropriate arguments for functions which depend on total ordering (for example, `min()`, `max()`, and `sorted()` produce undefined results given a list of sets as inputs).

  Comparison of sets enforces reflexivity of its elements.

- Most other built-in types have no comparison methods implemented, so they inherit the default comparison behavior.

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

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

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

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

  `x == y` 和 `y == x`

  `x != y` 和 `y != x`

  `x < y` 和 `y > x`

  `x <= y` 和 `y >= x`

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

  `x > y and y > z` 暗示了 `x > z`

  `x < y and y <= z` 暗示了 `x < z`

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

  `x == y` 和 `not x != y`

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

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

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

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

Python does not enforce these consistency rules. In fact, the not-a-number values are an example for not following these rules.

6.10.2 Membership test operations

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

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

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

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

Lastly, the old-style iteration protocol is tried: if a class defines *__getitem__()*, x in y is True if and only if there is a non-negative integer index *i* such that x is y[i] or x == y[i], and no lower integer index raises the IndexError exception. (If any other exception is raised, it is as if *in* raised that exception).

The operator *not in* is defined to have the inverse truth value of *in*.

6.10.3 Identity comparisons

The operators *is* and *is not* test for an object's identity: x is y is true if and only if *x* and *y* are the same object. An Object's identity is determined using the id() function. x is not y yields the inverse truth value.[4]

6.11 Boolean operations

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

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: False, None, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. User-defined objects can customize their truth value by providing a *__bool__()* method.

The operator *not* yields True if its argument is false, False otherwise.

The expression x and y first evaluates *x*; if *x* is false, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

The expression x or y first evaluates *x*; if *x* is true, its value is returned; otherwise, *y* is evaluated and the resulting value is returned.

Note that neither *and* nor *or* restrict the value and type they return to False and True, but rather return the last evaluated argument. This is sometimes useful, e.g., if s is a string that should be replaced by a default value if it is empty, the expression s or 'foo' yields the desired value. Because *not* has to create a new value, it returns a boolean value regardless of the type of its argument (for example, not 'foo' produces False rather than ''.)

6.12 Assignment expressions

```
assignment_expression ::= [identifier ":="] expression
```

An assignment expression (sometimes also called a "named expression" or "walrus") assigns an *expression* to an *identifier*, while also returning the value of the *expression*.

One common use case is when handling matched regular expressions:

```
if matching := pattern.search(data):
    do_something(matching)
```

Or, when processing a file stream in chunks:

```
while chunk := file.read(9000):
    process(chunk)
```

---

[4] Due to automatic garbage-collection, free lists, and the dynamic nature of descriptors, you may notice seemingly unusual behaviour in certain uses of the *is* operator, like those involving comparisons between instance methods, or constants. Check their documentation for more info.

Assignment expressions must be surrounded by parentheses when used as expression statements and when used as sub-expressions in slicing, conditional, lambda, keyword-argument, and comprehension-if expressions and in `assert`, `with`, and `assignment` statements. In all other places where they can be used, parentheses are not required, including in `if` and `while` statements.

在 3.8 版被加入: See **PEP 572** for more details about assignment expressions.

6.13 Conditional expressions

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

Conditional expressions (sometimes called a "ternary operator") have the lowest priority of all Python operations.

The expression `x if C else y` first evaluates the condition, *C* rather than *x*. If *C* is true, *x* is evaluated and its value is returned; otherwise, *y* is evaluated and its value is returned.

See **PEP 308** for more details about conditional expressions.

6.14 Lambdas

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

Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. The expression `lambda parameters: expression` yields a function object. The unnamed object behaves like a function object defined with:

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

See section 函式定義 for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements or annotations.

6.15 Expression lists

```
starred_expression        ::= ["*"] or_expr
flexible_expression        ::= assignment_expression | starred_expression
flexible_expression_list ::= flexible_expression ("," flexible_expression)* [","]
starred_expression_list  ::= starred_expression ("," starred_expression)* [","]
expression_list           ::= expression ("," expression)* [","]
yield_list                ::= expression_list | starred_expression "," [starred_expression_list]
```

Except when part of a list or set display, an expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

An asterisk `*` denotes *iterable unpacking*. Its operand must be an *iterable*. The iterable is expanded into a sequence of items, which are included in the new tuple, list, or set, at the site of the unpacking.

在 3.5 版被加入: Iterable unpacking in expression lists, originally proposed by **PEP 448**.

在 3.11 版被加入: Any item in an expression list may be starred. See **PEP 646**.

A trailing comma is required only to create a one-item tuple, such as `1,`; it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: `()`.)

6.16 Evaluation order

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

In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

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

6.17 Operator precedence

The following table summarizes the operator precedence in Python, from highest precedence (most binding) to lowest precedence (least binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation and conditional expressions, which group from right to left).

Note that comparisons, membership tests, and identity tests, all have the same precedence and have a left-to-right chaining feature as described in the *Comparisons* section.

運算子	描述
`(expressions...)`, `[expressions...]`, `{key: value...}`, `{expressions...}`	Binding or parenthesized expression, list display, dictionary display, set display
`x[index]`, `x[index:index]`, `x(arguments...)`, `x.attribute`	Subscription, slicing, call, attribute reference
`await x`	Await expression
`**`	Exponentiation[5]
`+x`, `-x`, `~x`	Positive, negative, bitwise NOT
`*`, `@`, `/`, `//`, `%`	Multiplication, matrix multiplication, division, floor division, remainder[6]
`+`, `-`	Addition and subtraction
`<<`, `>>`	Shifts
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `!=`, `==`	Comparisons, including membership tests and identity tests
`not x`	Boolean NOT
`and`	Boolean AND
`or`	Boolean OR
`if` -- `else`	Conditional expression
`lambda`	Lambda expression
`:=`	Assignment expression

⟨F⟩解

---

[5] The power operator `**` binds less tightly than an arithmetic or bitwise unary operator on its right, that is, `2**-1` is `0.5`.

[6] The `%` operator is also used for string formatting; the same precedence applies.

簡單陳述式

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

```
simple_stmt ::= expression_stmt
              | assert_stmt
              | assignment_stmt
              | augmented_assignment_stmt
              | annotated_assignment_stmt
              | pass_stmt
              | del_stmt
              | return_stmt
              | yield_stmt
              | raise_stmt
              | break_stmt
              | continue_stmt
              | import_stmt
              | future_stmt
              | global_stmt
              | nonlocal_stmt
              | type_stmt
```

7.1 Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value `None`). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

```
expression_stmt ::= starred_expression
```

An expression statement evaluates the expression list (which may be a single expression).

In interactive mode, if the value is not `None`, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output on a line by itself (except if the result is `None`, so that procedure calls do not cause any output.)

7.2 Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

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

(See section *Primaries* for the syntax definitions for *attributeref*, *subscription*, and *slicing*.)

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section 標準型⊞階層).

Assignment of an object to a target list, optionally enclosed in parentheses or square brackets, is recursively defined as follows.

- If the target list is a single target with no trailing comma, optionally in parentheses, the object is assigned to that target.

- Else:

  - If the target list contains one target prefixed with an asterisk, called a "starred" target: The object must be an iterable with at least as many items as there are targets in the target list, minus one. The first items of the iterable are assigned, from left to right, to the targets before the starred target. The final items of the iterable are assigned to the targets after the starred target. A list of the remaining items in the iterable is then assigned to the starred target (the list can be empty).

  - Else: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.

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

- If the target is an identifier (name):

  - If the name does not occur in a *global* or *nonlocal* statement in the current code block: the name is bound to the object in the current local namespace.

  - Otherwise: the name is bound to the object in the global namespace or the outer namespace determined by *nonlocal*, respectively.

  The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

- If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, `TypeError` is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily `AttributeError`).

  Note: If the object is a class instance and the attribute reference occurs on both sides of the assignment operator, the right-hand side expression, `a.x` can access either an instance attribute or (if no instance attribute exists) a class attribute. The left-hand side target `a.x` is always set as an instance attribute, creating it if necessary. Thus, the two occurrences of `a.x` do not necessarily refer to the same attribute: if the right-hand side expression refers to a class attribute, the left-hand side creates a new instance attribute as the target of the assignment:

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

This description does not necessarily apply to descriptor attributes, such as properties created with `property()`.

- If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

  If the primary is a mutable sequence object (such as a list), the subscript must yield an integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, `IndexError` is raised (assignment to a subscripted sequence cannot add new items to a list).

  If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/value pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

  For user-defined objects, the *__setitem__()* method is called with appropriate arguments.

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

CPython 實作細節： In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.

Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are 'simultaneous' (for example `a, b = b, a` swaps two variables), overlaps *within* the collection of assigned-to variables occur left-to-right, sometimes resulting in confusion. For instance, the following program prints `[0, 2]`:

```
x = [0, 1]
i = 0
i, x[i] = 1, 2          # i is updated, then x[i] is updated
print(x)
```

> ➔ **也參考**
>
> **PEP 3132 - Extended Iterable Unpacking**
>   The specification for the `*target` feature.

7.2.1 Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

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

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

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

An augmented assignment statement like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

Unlike normal assignments, augmented assignments evaluate the left-hand side *before* evaluating the right-hand side. For example, `a[i] += f(x)` first looks-up `a[i]`, then it evaluates `f(x)` and performs the addition, and lastly, it writes the result back to `a[i]`.

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented assignment statements is handled the same way as normal assignments. Similarly, with the exception of the possible *in-place* behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

For targets which are attribute references, the same *caveat about class and instance attributes* applies as for regular assignments.

7.2.2 Annotated assignment statements

Annotation assignment is the combination, in a single statement, of a variable or attribute annotation and an optional assignment statement:

```
annotated_assignment_stmt ::= augtarget ":" expression
                              ["=" (starred_expression | yield_expression)]
```

The difference from normal *Assignment statements* is that only a single target is allowed.

The assignment target is considered "simple" if it consists of a single name that is not enclosed in parentheses. For simple assignment targets, if in class or module scope, the annotations are evaluated and stored in a special class or module attribute `__annotations__` that is a dictionary mapping from variable names (mangled if private) to evaluated annotations. This attribute is writable and is automatically created at the start of class or module body execution, if annotations are found statically.

If the assignment target is not simple (an attribute, subscript node, or parenthesized name), the annotation is evaluated if in class or module scope, but not stored.

If a name is annotated in a function scope, then this name is local for that scope. Annotations are never evaluated and stored in function scopes.

If the right hand side is present, an annotated assignment performs the actual assignment before evaluating annotations (where applicable). If the right hand side is not present for an expression target, then the interpreter evaluates the target except for the last `__setitem__()` or `__setattr__()` call.

> **→ 也參考**
>
> **PEP 526** - **Syntax for Variable Annotations**
>   The proposal that added syntax for annotating the types of variables (including class variables and instance variables), instead of expressing them through comments.
>
> **PEP 484** - **Type hints**
>   The proposal that added the `typing` module to provide a standard syntax for type annotations that can be used in static analysis tools and IDEs.

在 3.8 版的變更: Now annotated assignments allow the same expressions in the right hand side as regular assignments. Previously, some expressions (like un-parenthesized tuple expressions) caused a syntax error.

7.3 `assert` 陳述式

Assert statements are a convenient way to insert debugging assertions into a program:

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

The simple form, `assert expression`, is equivalent to

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

The extended form, `assert expression1, expression2`, is equivalent to

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

These equivalences assume that `__debug__` and `AssertionError` refer to the built-in variables with those names. In the current implementation, the built-in variable `__debug__` is `True` under normal circumstances, `False` when optimization is requested (command line option `-O`). The current code generator emits no code for an *assert* statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to `__debug__` are illegal. The value for the built-in variable is determined when the interpreter starts.

7.4 `pass` 陳述式

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

pass is a null operation --- when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

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

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

7.5 `del` 陳述式

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

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

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

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

在 3.2 版的變更: Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

7.6 `return` 陳述式

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

return may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else `None` is substituted.

`return` leaves the current function call with the expression list (or `None`) as return value.

When `return` passes control out of a `try` statement with a `finally` clause, that `finally` clause is executed before really leaving the function.

In a generator function, the `return` statement indicates that the generator is done and will cause `StopIteration` to be raised. The returned value (if any) is used as an argument to construct `StopIteration` and becomes the `StopIteration.value` attribute.

In an asynchronous generator function, an empty `return` statement indicates that the asynchronous generator is done and will cause `StopAsyncIteration` to be raised. A non-empty `return` statement is a syntax error in an asynchronous generator function.

7.7 `yield` 陳述式

```
yield_stmt ::= yield_expression
```

A `yield` statement is semantically equivalent to a *yield expression*. The `yield` statement can be used to omit the parentheses that would otherwise be required in the equivalent yield expression statement. For example, the yield statements

```
yield <expr>
yield from <expr>
```

are equivalent to the yield expression statements

```
(yield <expr>)
(yield from <expr>)
```

Yield expressions and statements are only used when defining a *generator* function, and are only used in the body of the generator function. Using `yield` in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

For full details of `yield` semantics, refer to the *Yield expressions* section.

7.8 `raise` 陳述式

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

If no expressions are present, `raise` re-raises the exception that is currently being handled, which is also known as the *active exception*. If there isn't currently an active exception, a `RuntimeError` exception is raised indicating that this is an error.

Otherwise, `raise` evaluates the first expression as the exception object. It must be either a subclass or an instance of `BaseException`. If it is a class, the exception instance will be obtained when needed by instantiating the class with no arguments.

The *type* of the exception is the exception instance's class, the *value* is the instance itself.

A traceback object is normally created automatically when an exception is raised and attached to it as the `__traceback__` attribute. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

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

The `from` clause is used for exception chaining: if given, the second *expression* must be another exception class or instance. If the second expression is an exception instance, it will be attached to the raised exception as the `__cause__` attribute (which is writable). If the expression is an exception class, the class will be instantiated and the resulting exception instance will be attached to the raised exception as the `__cause__` attribute. If the raised exception is not handled, both exceptions will be printed:

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

The above exception was the direct cause of the following exception:

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

A similar mechanism works implicitly if a new exception is raised when an exception is already being handled. An exception may be handled when an *except* or *finally* clause, or a *with* statement, is used. The previous exception is then attached as the new exception's __context__ attribute:

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

During handling of the above exception, another exception occurred:

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

Exception chaining can be explicitly suppressed by specifying None in the from clause:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Additional information on exceptions can be found in section 例外, and information about handling exceptions is in section *try 陳述式*.

在 3.3 版的變更: None is now permitted as Y in raise X from Y.

Added the __suppress_context__ attribute to suppress automatic display of the exception context.

在 3.11 版的變更: If the traceback of the active exception is modified in an *except* clause, a subsequent raise statement re-raises the exception with the modified traceback. Previously, the exception was re-raised with the traceback it had when it was caught.

7.9 `break` 陳述式

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

break may only occur syntactically nested in a *for* or *while* loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional `else` clause if the loop has one.

If a *for* loop is terminated by *break*, the loop control target keeps its current value.

When *break* passes control out of a *try* statement with a *finally* clause, that `finally` clause is executed before really leaving the loop.

7.10 `continue` 陳述式

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

continue may only occur syntactically nested in a *for* or *while* loop, but not nested in a function or class definition within that loop. It continues with the next cycle of the nearest enclosing loop.

When *continue* passes control out of a *try* statement with a *finally* clause, that `finally` clause is executed before really starting the next loop cycle.

7.11 `import` 陳述式

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

The basic import statement (no *from* clause) is executed in two steps:

1. find a module, loading and initializing it if necessary

2. define a name or names in the local namespace for the scope where the *import* statement occurs.

When the statement contains multiple clauses (separated by commas) the two steps are carried out separately for each clause, just as though the clauses had been separated out into individual import statements.

The details of the first step, finding and loading modules, are described in greater detail in the section on the *import system*, which also describes the various types of packages and modules that can be imported, as well as all the hooks that can be used to customize the import system. Note that failures in this step may indicate either that the module could not be located, *or* that an error occurred while initializing the module, which includes execution of the module's code.

If the requested module is retrieved successfully, it will be made available in the local namespace in one of three ways:

- If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

- If no other name is specified, and the module being imported is a top level module, the module's name is bound in the local namespace as a reference to the imported module

- If the module being imported is *not* a top level module, then the name of the top level package that contains the module is bound in the local namespace as a reference to the top level package. The imported module must be accessed using its full qualified name rather than directly

The *from* form uses a slightly more complex process:

1. find the module specified in the *from* clause, loading and initializing it if necessary;

2. for each of the identifiers specified in the *import* clauses:

   1. check if the imported module has an attribute by that name

   2. if not, attempt to import a submodule with that name and then check the imported module again for that attribute

   3. if the attribute is not found, `ImportError` is raised.

   4. otherwise, a reference to that value is stored in the local namespace, using the name in the `as` clause if it is present, otherwise using the attribute name

範例:

```python
import foo                 # foo imported and bound locally
import foo.bar.baz         # foo, foo.bar, and foo.bar.baz imported, foo bound locally
import foo.bar.baz as fbb  # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz bound as fbb
from foo.bar import baz    # foo, foo.bar, and foo.bar.baz imported, foo.bar.baz bound as baz
from foo import attr       # foo imported and foo.attr bound as attr
```

If the list of identifiers is replaced by a star (`'*'`), all public names defined in the module are bound in the local namespace for the scope where the *import* statement occurs.

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

The wild card form of import --- `from module import *` --- is only allowed at the module level. Attempting to use it in class or function definitions will raise a `SyntaxError`.

When specifying what module to import you do not have to specify the absolute name of the module. When a module or package is contained within another package it is possible to make a relative import within the same top package without having to mention the package name. By using leading dots in the specified module or package after *from* you can specify how high to traverse up the current package hierarchy without specifying exact names. One leading dot means the current package where the module making the import exists. Two dots means up one package level. Three dots is up two levels, etc. So if you execute `from . import mod` from a module in the `pkg` package then you will end up importing `pkg.mod`. If you execute `from ..subpkg2 import mod` from within `pkg.subpkg1` you will import `pkg.subpkg2.mod`. The specification for relative imports is contained in the *Package Relative Imports* section.

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

引發一個附帶引數 `module`、`filename`、`sys.path`、`sys.meta_path`、`sys.path_hooks` 的稽核事件 `import`。

7.11.1 Future statements

A *future statement* is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python where the feature becomes standard.

The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

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

A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

- the module docstring (if any),

- comments,

- blank lines, and

- other future statements.

The only feature that requires using the future statement is `annotations` (see **PEP 563**).

All historical features enabled by the future statement are still recognized by Python 3. The list includes `absolute_import`, `division`, `generators`, `generator_stop`, `unicode_literals`, `print_function`, `nested_scopes` and `with_statement`. They are all redundant because they are always enabled, and only kept for backwards compatibility.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The direct runtime semantics are the same as for any import statement: there is a standard module `__future__`, described later, and it will be imported in the usual way at the time the future statement is executed.

The interesting runtime semantics depend on the specific feature enabled by the future statement.

Note that there is nothing special about the statement:

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

That is not a future statement; it's an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by calls to the built-in functions `exec()` and `compile()` that occur in a module M containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can be controlled by optional arguments to `compile()` --- see the documentation of that function for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the `-i` option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.

> ↪ **也參考**
>
> **PEP 236** - **Back to the __future__**
>   The original proposal for the __future__ mechanism.

7.12 `global` 陳述式

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

The `global` statement causes the listed identifiers to be interpreted as globals. It would be impossible to assign to a global variable without `global`, although free variables may refer to globals without being declared global.

The `global` statement applies to the entire scope of a function or class body. A `SyntaxError` is raised if a variable is used or assigned to prior to its global declaration in the scope.

Programmer's note: `global` is a directive to the parser. It applies only to code parsed at the same time as the `global` statement. In particular, a `global` statement contained in a string or code object supplied to the built-in `exec()` function does not affect the code block *containing* the function call, and code contained in such a string

is unaffected by `global` statements in the code containing the function call. The same applies to the `eval()` and `compile()` functions.

7.13 `nonlocal` 陳述式

```
nonlocal_stmt ::= "nonlocal" identifier ("," identifier)*
```

When the definition of a function or class is nested (enclosed) within the definitions of other functions, its nonlocal scopes are the local scopes of the enclosing functions. The *nonlocal* statement causes the listed identifiers to refer to names previously bound in nonlocal scopes. It allows encapsulated code to rebind such nonlocal identifiers. If a name is bound in more than one nonlocal scope, the nearest binding is used. If a name is not bound in any nonlocal scope, or if there is no nonlocal scope, a `SyntaxError` is raised.

The *nonlocal* statement applies to the entire scope of a function or class body. A `SyntaxError` is raised if a variable is used or assigned to prior to its nonlocal declaration in the scope.

> **↪ 也參考**
>
> **PEP 3104 - Access to Names in Outer Scopes**
>   The specification for the *nonlocal* statement.

Programmer's note: *nonlocal* is a directive to the parser and applies only to code parsed along with it. See the note for the *global* statement.

7.14 `type` 陳述式

```
type_stmt ::= 'type' identifier [type_params] "=" expression
```

The `type` statement declares a type alias, which is an instance of `typing.TypeAliasType`.

For example, the following statement creates a type alias:

```
type Point = tuple[float, float]
```

This code is roughly equivalent to:

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

annotation-def indicates an *annotation scope*, which behaves mostly like a function, but with several small differences.

The value of the type alias is evaluated in the annotation scope. It is not evaluated when the type alias is created, but only when the value is accessed through the type alias's `__value__` attribute (see *Lazy evaluation*). This allows the type alias to refer to names that are not yet defined.

Type aliases may be made generic by adding a *type parameter list* after the name. See *Generic type aliases* for more.

`type` is a *soft keyword*.

在 3.12 版被加入.

> **↪ 也參考**
>
> **PEP 695 - Type Parameter Syntax**
>   Introduced the `type` statement and syntax for generic classes and functions.

CHAPTER 8

𝔽合陳述式

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.

The `if`, `while` and `for` statements implement traditional control flow constructs. `try` specifies exception handlers and/or cleanup code for a group of statements, while the `with` statement allows the execution of initialization and finalization code around a block of code. Function and class definitions are also syntactically compound statements.

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

```
if test1: if test2: print(x)
```

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

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

Summarizing:

```
compound_stmt ::= if_stmt
                | while_stmt
                | for_stmt
                | try_stmt
                | with_stmt
                | match_stmt
                | funcdef
                | classdef
                | async_with_stmt
                | async_for_stmt
                | async_funcdef
suite         ::= stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement     ::= stmt_list NEWLINE | compound_stmt
```

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

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

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.

8.1 `if` 陳述式

The *if* statement is used for conditional execution:

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

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section *Boolean operations* for the definition of true and false); then that suite is executed (and no other part of the *if* statement is executed or evaluated). If all expressions are false, the suite of the *else* clause, if present, is executed.

8.2 `while` 陳述式

The *while* statement is used for repeated execution as long as an expression is true:

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

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the else clause, if present, is executed and the loop terminates.

A *break* statement executed in the first suite terminates the loop without executing the else clause's suite. A *continue* statement executed in the first suite skips the rest of the suite and goes back to testing the expression.

8.3 `for` 陳述式

The *for* statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

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

The starred_list expression is evaluated once; it should yield an *iterable* object. An *iterator* is created for that iterable. The first item provided by the iterator is then assigned to the target list using the standard rules for assignments (see *Assignment statements*), and the suite is executed. This repeats for each item provided by the iterator. When the iterator is exhausted, the suite in the else clause, if present, is executed, and the loop terminates.

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

The for-loop makes assignments to the variables in the target list. This overwrites all previous assignments to those variables including those made in the suite of the for-loop:

```
for i in range(10):
    print(i)
    i = 5              # this will not affect the for-loop
                       # because i will be overwritten with the next
                       # index in the range
```

Names in the target list are not deleted when the loop is finished, but if the sequence is empty, they will not have been assigned to at all by the loop. Hint: the built-in type `range()` represents immutable arithmetic sequences of integers. For instance, iterating `range(3)` successively yields 0, 1, and then 2.

在 3.11 版的變更: Starred elements are now allowed in the expression list.

8.4 `try` 陳述式

The `try` statement specifies exception handlers and/or cleanup code for a group of statements:

```
try_stmt  ::= try1_stmt | try2_stmt | try3_stmt
try1_stmt ::= "try" ":" suite
              ("except" [expression ["as" identifier]] ":" suite)+
              ["else" ":" suite]
              ["finally" ":" suite]
try2_stmt ::= "try" ":" suite
              ("except" "*" expression ["as" identifier] ":" suite)+
              ["else" ":" suite]
              ["finally" ":" suite]
try3_stmt ::= "try" ":" suite
              "finally" ":" suite
```

Additional information on exceptions can be found in section 例外, and information on using the `raise` statement to generate exceptions may be found in section *raise 陳述式*.

8.4.1 `except` 子句

The `except` clause(s) specify one or more exception handlers. When no exception occurs in the `try` clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the `except` clauses in turn until one is found that matches the exception. An expression-less `except` clause, if present, must be last; it matches any exception.

For an `except` clause with an expression, the expression must evaluate to an exception type or a tuple of exception types. The raised exception matches an `except` clause whose expression evaluates to the class or a *non-virtual base class* of the exception object, or to a tuple that contains such a class.

If no `except` clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack.[1]

If the evaluation of an expression in the header of an `except` clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire `try` statement raised the exception).

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

When an exception has been assigned using `as target`, it is cleared at the end of the `except` clause. This is as if

```
except E as N:
    foo
```

was translated to

```
except E as N:
    try:
        foo
```

(繼續下一頁)

---

[1] The exception is propagated to the invocation stack unless there is a `finally` clause which happens to raise another exception. That new exception causes the old one to be lost.

```
    finally:
        del N
```

This means the exception must be assigned to a different name to be able to refer to it after the `except` clause. Exceptions are cleared because with the traceback attached to them, they form a reference cycle with the stack frame, keeping all locals in that frame alive until the next garbage collection occurs.

Before an `except` clause's suite is executed, the exception is stored in the `sys` module, where it can be accessed from within the body of the `except` clause by calling `sys.exception()`. When leaving an exception handler, the exception stored in the `sys` module is reset to its previous value:

```
>>> print(sys.exception())
None
>>> try:
...     raise TypeError
... except:
...     print(repr(sys.exception()))
...     try:
...         raise ValueError
...     except:
...         print(repr(sys.exception()))
...     print(repr(sys.exception()))
...
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None
```

8.4.2 `except*` 子句

The `except*` clause(s) are used for handling `ExceptionGroups`. The exception type for matching is interpreted as in the case of *except*, but in the case of exception groups we can have partial matches when the type matches some of the exceptions in the group. This means that multiple `except*` clauses can execute, each handling part of the exception group. Each clause executes at most once and handles an exception group of all matching exceptions. Each exception in the group is handled by at most one `except*` clause, the first that matches it.

```
>>> try:
...     raise ExceptionGroup("eg",
...         [ValueError(1), TypeError(2), OSError(3), OSError(4)])
... except* TypeError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
... except* OSError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
...
caught <class 'ExceptionGroup'> with nested (TypeError(2),)
caught <class 'ExceptionGroup'> with nested (OSError(3), OSError(4))
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  | ExceptionGroup: eg
  +-+--------------- 1 ----------------
    | ValueError: 1
    +----------------------------------
```

Any remaining exceptions that were not handled by any `except*` clause are re-raised at the end, along with all exceptions that were raised from within the `except*` clauses. If this list contains more than one exception to reraise, they are combined into an exception group.

If the raised exception is not an exception group and its type matches one of the `except*` clauses, it is caught and wrapped by an exception group with an empty message string.

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

An `except*` clause must have a matching expression; it cannot be `except*:`. Furthermore, this expression cannot contain exception group types, because that would have ambiguous semantics.

It is not possible to mix `except` and `except*` in the same `try`. `break`, `continue` and `return` cannot appear in an `except*` clause.

8.4.3 `else` 子句

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

8.4.4 `finally` 子句

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

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

The exception information is not available to the program during execution of the `finally` clause.

When a `return`, `break` or `continue` statement is executed in the `try` suite of a try...finally statement, the `finally` clause is also executed 'on the way out.'

The return value of a function is determined by the last `return` statement executed. Since the `finally` clause always executes, a `return` statement executed in the `finally` clause will always be the last one executed:

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

在 3.8 版的變更: Prior to Python 3.8, a `continue` statement was illegal in the `finally` clause due to a problem with the implementation.

8.5 `with` 陳述式

The `with` statement is used to wrap the execution of a block with methods defined by a context manager (see section *With 陳述式的情境管理器*). This allows common *try...except...finally* usage patterns to be encapsulated for convenient reuse.

```
with_stmt          ::= "with" ( "(" with_stmt_contents ","? ")" | with_stmt_contents ) ":" suite
with_stmt_contents ::= with_item ("," with_item)*
with_item          ::= expression ["as" target]
```

The execution of the `with` statement with one "item" proceeds as follows:

1. The context expression (the expression given in the `with_item`) is evaluated to obtain a context manager.

2. The context manager's `__enter__()` is loaded for later use.

3. The context manager's `__exit__()` is loaded for later use.

4. The context manager's `__enter__()` method is invoked.

5. If a target was included in the `with` statement, the return value from `__enter__()` is assigned to it.

> 🛈 **備**𝔼
>
> The `with` statement guarantees that if the `__enter__()` method returns without an error, then `__exit__()` will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 7 below.

6. The suite is executed.

7. The context manager's `__exit__()` method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to `__exit__()`. Otherwise, three `None` arguments are supplied.

   If the suite was exited due to an exception, and the return value from the `__exit__()` method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the `with` statement.

   If the suite was exited for any reason other than an exception, the return value from `__exit__()` is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

以下程式碼：

```
with EXPRESSION as TARGET:
    SUITE
```

在語義上等同於：

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

With more than one item, the context managers are processed as if multiple `with` statements were nested:

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

在語義上等同於：

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

You can also write multi-item context managers in multiple lines if the items are surrounded by parentheses. For example:

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

在 3.1 版的變更: Support for multiple context expressions.

在 3.10 版的變更: Support for using grouping parentheses to break the statement in multiple lines.

> **↪ 也參考**
>
> **PEP 343 - The "with" statement**
> The specification, background, and examples for the Python `with` statement.

8.6 `match` 陳述式

在 3.10 版被加入.

The match statement is used for pattern matching. Syntax:

```
match_stmt   ::= 'match' subject_expr ":" NEWLINE INDENT case_block+ DEDENT
subject_expr ::= star_named_expression "," star_named_expressions?
                 | named_expression
case_block   ::= 'case' patterns [guard] ":" block
```

> **ⓘ 備註**
>
> This section uses single quotes to denote *soft keywords*.

Pattern matching takes a pattern as input (following `case`) and a subject value (following `match`). The pattern (which may contain subpatterns) is matched against the subject value. The outcomes are:

- A match success or failure (also termed a pattern success or failure).

- Possible binding of matched values to a name. The prerequisites for this are further discussed below.

The `match` and `case` keywords are *soft keywords*.

> **↪ 也參考**
>
> - **PEP 634** -- Structural Pattern Matching: Specification
>
> - **PEP 636** -- Structural Pattern Matching: Tutorial

8.6.1 Overview

Here's an overview of the logical flow of a match statement:

1. The subject expression `subject_expr` is evaluated and a resulting subject value obtained. If the subject expression contains a comma, a tuple is constructed using the standard rules.

2. Each pattern in a `case_block` is attempted to match with the subject value. The specific rules for success or failure are described below. The match attempt can also bind some or all of the standalone names within the pattern. The precise pattern binding rules vary per pattern type and are specified below. **Name bindings made during a successful pattern match outlive the executed block and can be used after the match statement**.

> ℹ️ **備⿕**
>
> During failed pattern matches, some subpatterns may succeed. Do not rely on bindings being made for a failed match. Conversely, do not rely on variables remaining unchanged after a failed match. The exact behavior is dependent on implementation and may vary. This is an intentional decision made to allow different implementations to add optimizations.

3. If the pattern succeeds, the corresponding guard (if present) is evaluated. In this case all name bindings are guaranteed to have happened.

   - If the guard evaluates as true or is missing, the `block` inside `case_block` is executed.

   - Otherwise, the next `case_block` is attempted as described above.

   - If there are no further case blocks, the match statement is completed.

> ℹ️ **備⿕**
>
> Users should generally never rely on a pattern being evaluated. Depending on implementation, the interpreter may cache values or use other optimizations which skip repeated evaluations.

A sample match statement:

```
>>> flag = False
>>> match (100, 200):
...     case (100, 300):  # Mismatch: 200 != 300
...         print('Case 1')
...     case (100, 200) if flag:  # Successful match, but guard fails
...         print('Case 2')
...     case (100, y):  # Matches and binds y to 200
...         print(f'Case 3, y: {y}')
...     case _:  # Pattern not attempted
...         print('Case 4, I match anything!')
...
Case 3, y: 200
```

In this case, `if flag` is a guard. Read more about that in the next section.

8.6.2 Guards

```
guard ::= "if" named_expression
```

A `guard` (which is part of the `case`) must succeed for code inside the `case` block to execute. It takes the form: *if* followed by an expression.

The logical flow of a `case` block with a `guard` follows:

1. Check that the pattern in the `case` block succeeded. If the pattern failed, the `guard` is not evaluated and the next `case` block is checked.

2. If the pattern succeeded, evaluate the `guard`.

   - If the `guard` condition evaluates as true, the case block is selected.

   - If the `guard` condition evaluates as false, the case block is not selected.

- If the `guard` raises an exception during evaluation, the exception bubbles up.

Guards are allowed to have side effects as they are expressions. Guard evaluation must proceed from the first to the last case block, one at a time, skipping case blocks whose pattern(s) don't all succeed. (I.e., guard evaluation must happen in order.) Guard evaluation must stop once a case block is selected.

8.6.3 Irrefutable Case Blocks

An irrefutable case block is a match-all case block. A match statement may have at most one irrefutable case block, and it must be last.

A case block is considered irrefutable if it has no guard and its pattern is irrefutable. A pattern is considered irrefutable if we can prove from its syntax alone that it will always succeed. Only the following patterns are irrefutable:

- *AS Patterns* whose left-hand side is irrefutable
- *OR Patterns* containing at least one irrefutable pattern
- *Capture Patterns*
- *Wildcard Patterns*
- parenthesized irrefutable patterns

8.6.4 Patterns

> ℹ️ **備㊕**
>
> This section uses grammar notations beyond standard EBNF:
>
> - the notation `SEP.RULE+` is shorthand for `RULE (SEP RULE)*`
> - the notation `!RULE` is shorthand for a negative lookahead assertion

The top-level syntax for `patterns` is:

```
patterns       ::= open_sequence_pattern | pattern
pattern        ::= as_pattern | or_pattern
closed_pattern ::= | literal_pattern
                   | capture_pattern
                   | wildcard_pattern
                   | value_pattern
                   | group_pattern
                   | sequence_pattern
                   | mapping_pattern
                   | class_pattern
```

The descriptions below will include a description "in simple terms" of what a pattern does for illustration purposes (credits to Raymond Hettinger for a document that inspired most of the descriptions). Note that these descriptions are purely for illustration purposes and **may not** reflect the underlying implementation. Furthermore, they do not cover all valid forms.

OR Patterns

An OR pattern is two or more patterns separated by vertical bars `|`. Syntax:

```
or_pattern ::= "|".closed_pattern+
```

Only the final subpattern may be *irrefutable*, and each subpattern must bind the same set of names to avoid ambiguity.

An OR pattern matches each of its subpatterns in turn to the subject value, until one succeeds. The OR pattern is then considered successful. Otherwise, if none of the subpatterns succeed, the OR pattern fails.

In simple terms, `P1 | P2 | ...` will try to match `P1`, if it fails it will try to match `P2`, succeeding immediately if any succeeds, failing otherwise.

AS Patterns

An AS pattern matches an OR pattern on the left of the `as` keyword against a subject. Syntax:

```
as_pattern ::= or_pattern "as" capture_pattern
```

If the OR pattern fails, the AS pattern fails. Otherwise, the AS pattern binds the subject to the name on the right of the as keyword and succeeds. `capture_pattern` cannot be a `_`.

In simple terms `P as NAME` will match with `P`, and on success it will set `NAME = <subject>`.

Literal Patterns

A literal pattern corresponds to most *literals* in Python. Syntax:

```
literal_pattern ::= signed_number
                  | signed_number "+" NUMBER
                  | signed_number "-" NUMBER
                  | strings
                  | "None"
                  | "True"
                  | "False"
signed_number   ::= ["-"] NUMBER
```

The rule `strings` and the token `NUMBER` are defined in the *standard Python grammar*. Triple-quoted strings are supported. Raw strings and byte strings are supported. *f-string（f 字串）* are not supported.

The forms `signed_number '+' NUMBER` and `signed_number '-' NUMBER` are for expressing *complex numbers*; they require a real number on the left and an imaginary number on the right. E.g. `3 + 4j`.

In simple terms, `LITERAL` will succeed only if `<subject> == LITERAL`. For the singletons `None`, `True` and `False`, the `is` operator is used.

Capture Patterns

A capture pattern binds the subject value to a name. Syntax:

```
capture_pattern ::= !'_' NAME
```

A single underscore `_` is not a capture pattern (this is what `!'_'` expresses). It is instead treated as a *wildcard_pattern*.

In a given pattern, a given name can only be bound once. E.g. `case x, x: ...` is invalid while `case [x] | x: ...` is allowed.

Capture patterns always succeed. The binding follows scoping rules established by the assignment expression operator in **PEP 572**; the name becomes a local variable in the closest containing function scope unless there's an applicable *global* or *nonlocal* statement.

In simple terms `NAME` will always succeed and it will set `NAME = <subject>`.

Wildcard Patterns

A wildcard pattern always succeeds (matches anything) and binds no name. Syntax:

```
wildcard_pattern ::= '_'
```

`_` is a *soft keyword* within any pattern, but only within patterns. It is an identifier, as usual, even within `match` subject expressions, `guards`, and `case` blocks.

In simple terms, `_` will always succeed.

Value Patterns

A value pattern represents a named value in Python. Syntax:

```
value_pattern ::= attr
attr          ::= name_or_attr "." NAME
name_or_attr  ::= attr | NAME
```

The dotted name in the pattern is looked up using standard Python *name resolution rules*. The pattern succeeds if the value found compares equal to the subject value (using the == equality operator).

In simple terms NAME1.NAME2 will succeed only if `<subject> == NAME1.NAME2`

> **ℹ️ 備註**
>
> If the same value occurs multiple times in the same match statement, the interpreter may cache the first value found and reuse it rather than repeat the same lookup. This cache is strictly tied to a given execution of a given match statement.

Group Patterns

A group pattern allows users to add parentheses around patterns to emphasize the intended grouping. Otherwise, it has no additional syntax. Syntax:

```
group_pattern ::= "(" pattern ")"
```

In simple terms (P) has the same effect as P.

Sequence Patterns

A sequence pattern contains several subpatterns to be matched against sequence elements. The syntax is similar to the unpacking of a list or tuple.

```
sequence_pattern       ::= "[" [maybe_sequence_pattern] "]"
                         | "(" [open_sequence_pattern] ")"
open_sequence_pattern  ::= maybe_star_pattern "," [maybe_sequence_pattern]
maybe_sequence_pattern ::= ",".maybe_star_pattern+ ","?
maybe_star_pattern     ::= star_pattern | pattern
star_pattern           ::= "*" (capture_pattern | wildcard_pattern)
```

There is no difference if parentheses or square brackets are used for sequence patterns (i.e. (...) vs [...]).

> **ℹ️ 備註**
>
> A single pattern enclosed in parentheses without a trailing comma (e.g. (3 | 4)) is a *group pattern*. While a single pattern enclosed in square brackets (e.g. [3 | 4]) is still a sequence pattern.

At most one star subpattern may be in a sequence pattern. The star subpattern may occur in any position. If no star subpattern is present, the sequence pattern is a fixed-length sequence pattern; otherwise it is a variable-length sequence pattern.

The following is the logical flow for matching a sequence pattern against a subject value:

1. If the subject value is not a sequence[2], the sequence pattern fails.

---

[2] In pattern matching, a sequence is defined as one of the following:
- a class that inherits from `collections.abc.Sequence`
- a Python class that has been registered as `collections.abc.Sequence`
- a builtin class that has its (CPython) `Py_TPFLAGS_SEQUENCE` bit set
- a class that inherits from any of the above

The following standard library classes are sequences:
- `array.array`

2. If the subject value is an instance of `str`, `bytes` or `bytearray` the sequence pattern fails.

3. The subsequent steps depend on whether the sequence pattern is fixed or variable-length.

    If the sequence pattern is fixed-length:

    1. If the length of the subject sequence is not equal to the number of subpatterns, the sequence pattern fails

    2. Subpatterns in the sequence pattern are matched to their corresponding items in the subject sequence from left to right. Matching stops as soon as a subpattern fails. If all subpatterns succeed in matching their corresponding item, the sequence pattern succeeds.

    Otherwise, if the sequence pattern is variable-length:

    1. If the length of the subject sequence is less than the number of non-star subpatterns, the sequence pattern fails.

    2. The leading non-star subpatterns are matched to their corresponding items as for fixed-length sequences.

    3. If the previous step succeeds, the star subpattern matches a list formed of the remaining subject items, excluding the remaining items corresponding to non-star subpatterns following the star subpattern.

    4. Remaining non-star subpatterns are matched to their corresponding subject items, as for a fixed-length sequence.

> **ℹ 備註**
>
> The length of the subject sequence is obtained via `len()` (i.e. via the `__len__()` protocol). This length may be cached by the interpreter in a similar manner as *value patterns*.

In simple terms `[P1, P2, P3, ... , P<N>]` matches only if all the following happens:

- check `<subject>` is a sequence
- `len(subject) == <N>`
- `P1` matches `<subject>[0]` (note that this match can also bind names)
- `P2` matches `<subject>[1]` (note that this match can also bind names)
- ... and so on for the corresponding pattern/element.

Mapping Patterns

A mapping pattern contains one or more key-value patterns. The syntax is similar to the construction of a dictionary. Syntax:

```
mapping_pattern     ::= "{" [items_pattern] "}"
items_pattern       ::= ",".key_value_pattern+ ","?
key_value_pattern   ::= (literal_pattern | value_pattern) ":" pattern
                        | double_star_pattern
double_star_pattern ::= "**" capture_pattern
```

At most one double star pattern may be in a mapping pattern. The double star pattern must be the last subpattern in the mapping pattern.

- `collections.deque`
- `list`
- `memoryview`
- `range`
- `tuple`

> **ℹ 備註**
>
> Subject values of type `str`, `bytes`, and `bytearray` do not match sequence patterns.

Duplicate keys in mapping patterns are disallowed. Duplicate literal keys will raise a `SyntaxError`. Two keys that otherwise have the same value will raise a `ValueError` at runtime.

The following is the logical flow for matching a mapping pattern against a subject value:

1. If the subject value is not a mapping[3],the mapping pattern fails.

2. If every key given in the mapping pattern is present in the subject mapping, and the pattern for each key matches the corresponding item of the subject mapping, the mapping pattern succeeds.

3. If duplicate keys are detected in the mapping pattern, the pattern is considered invalid. A `SyntaxError` is raised for duplicate literal values; or a `ValueError` for named keys of the same value.

> **ℹ 備$\mathbb{F}$**
>
> Key-value pairs are matched using the two-argument form of the mapping subject's `get()` method. Matched key-value pairs must already be present in the mapping, and not created on-the-fly via `__missing__()` or `__getitem__()`.

In simple terms `{KEY1: P1, KEY2: P2, ... }` matches only if all the following happens:

- check `<subject>` is a mapping

- `KEY1 in <subject>`

- `P1` matches `<subject>[KEY1]`

- ... and so on for the corresponding KEY/pattern pair.

Class Patterns

A class pattern represents a class and its positional and keyword arguments (if any). Syntax:

```
class_pattern       ::= name_or_attr "(" [pattern_arguments ","?] ")"
pattern_arguments   ::= positional_patterns ["," keyword_patterns]
                        | keyword_patterns
positional_patterns ::= ",".pattern+
keyword_patterns    ::= ",".keyword_pattern+
keyword_pattern     ::= NAME "=" pattern
```

The same keyword should not be repeated in class patterns.

The following is the logical flow for matching a class pattern against a subject value:

1. If `name_or_attr` is not an instance of the builtin `type` , raise `TypeError`.

2. If the subject value is not an instance of `name_or_attr` (tested via `isinstance()`), the class pattern fails.

3. If no pattern arguments are present, the pattern succeeds. Otherwise, the subsequent steps depend on whether keyword or positional argument patterns are present.

    For a number of built-in types (specified below), a single positional subpattern is accepted which will match the entire subject; for these types keyword patterns also work as for other types.

    If only keyword patterns are present, they are processed as follows, one by one:

    I. The keyword is looked up as an attribute on the subject.

    - If this raises an exception other than `AttributeError`, the exception bubbles up.

    - If this raises `AttributeError`, the class pattern has failed.

---

[3] In pattern matching, a mapping is defined as one of the following:
- a class that inherits from `collections.abc.Mapping`
- a Python class that has been registered as `collections.abc.Mapping`
- a builtin class that has its (CPython) `Py_TPFLAGS_MAPPING` bit set
- a class that inherits from any of the above

The standard library classes `dict` and `types.MappingProxyType` are mappings.

- Else, the subpattern associated with the keyword pattern is matched against the subject's attribute value. If this fails, the class pattern fails; if this succeeds, the match proceeds to the next keyword.

II. If all keyword patterns succeed, the class pattern succeeds.

If any positional patterns are present, they are converted to keyword patterns using the *__match_args__* attribute on the class `name_or_attr` before matching:

I. The equivalent of `getattr(cls, "__match_args__", ())` is called.

- If this raises an exception, the exception bubbles up.

- If the returned value is not a tuple, the conversion fails and `TypeError` is raised.

- If there are more positional patterns than `len(cls.__match_args__)`, `TypeError` is raised.

- Otherwise, positional pattern `i` is converted to a keyword pattern using `__match_args__[i]` as the keyword. `__match_args__[i]` must be a string; if not `TypeError` is raised.

- If there are duplicate keywords, `TypeError` is raised.

> ↪ **也參考**
>
> *Customizing positional arguments in class pattern matching*

II. Once all positional patterns have been converted to keyword patterns,
the match proceeds as if there were only keyword patterns.

For the following built-in types the handling of positional subpatterns is different:

- `bool`
- `bytearray`
- `bytes`
- `dict`
- `float`
- `frozenset`
- `int`
- `list`
- `set`
- `str`
- `tuple`

These classes accept a single positional argument, and the pattern there is matched against the whole object rather than an attribute. For example `int(0|1)` matches the value `0`, but not the value `0.0`.

In simple terms `CLS(P1, attr=P2)` matches only if the following happens:

- `isinstance(<subject>, CLS)`

- convert `P1` to a keyword pattern using `CLS.__match_args__`

- For each keyword argument `attr=P2`:

  - `hasattr(<subject>, "attr")`

  - `P2` matches `<subject>.attr`

- ... and so on for the corresponding keyword argument/pattern pair.

8.7 函式定義

A function definition defines a user-defined function object (see section 標準型⬚階層):

```
funcdef                   ::= [decorators] "def" funcname [type_params] "(" [parameter_list] ")"
                              ["->" expression] ":" suite
decorators                ::= decorator+
decorator                 ::= "@" assignment_expression NEWLINE
parameter_list            ::= defparameter ("," defparameter)* "," "/" ["," [parameter_list_no_p
                              | parameter_list_no_posonly
parameter_list_no_posonly ::= defparameter ("," defparameter)* ["," [parameter_list_starargs]]
                              | parameter_list_starargs
parameter_list_starargs   ::= "*" [star_parameter] ("," defparameter)* ["," [parameter_star_kwar
                              "*" ("," defparameter)+ ["," [parameter_star_kwargs]]
                              | parameter_star_kwargs
parameter_star_kwargs     ::= "**" parameter [","]
parameter                 ::= identifier [":" expression]
star_parameter            ::= identifier [":" ["*"] expression]
defparameter              ::= parameter ["=" expression]
funcname                  ::= identifier
```

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called.[4]

A function definition may be wrapped by one or more *decorator* expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code

```
@f1(arg)
@f2
def func(): pass
```

大致等價於：

```
def func(): pass
func = f1(arg)(f2(func))
```

except that the original function is not temporarily bound to the name `func`.

在 3.9 版的變更: Functions may be decorated with any valid *assignment_expression*. Previously, the grammar was much more restrictive; see **PEP 614** for details.

A list of *type parameters* may be given in square brackets between the function's name and the opening parenthesis for its parameter list. This indicates to static type checkers that the function is generic. At runtime, the type parameters can be retrieved from the function's `__type_params__` attribute. See *Generic functions* for more.

在 3.12 版的變更: Type parameter lists are new in Python 3.12.

---

[4] A string literal appearing as the first statement in the function body is transformed into the function's `__doc__` attribute and therefore the function's *docstring*.

When one or more *parameters* have the form *parameter = expression*, the function is said to have "default parameter values." For a parameter with a default value, the corresponding *argument* may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters up until the "*" must also have a default value --- this is a syntactic restriction that is not expressed by the grammar.

Default parameter values are evaluated from left to right when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same "pre-computed" value is used for each call. This is especially important to understand when a default parameter value is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default parameter value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```python
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section *Calls*. A function call always assigns values to all parameters mentioned in the parameter list, either from positional arguments, from keyword arguments, or from default values. If the form "`*identifier`" is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form "`**identifier`" is present, it is initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type. Parameters after "*" or "`*identifier`" are keyword-only parameters and may only be passed by keyword arguments. Parameters before "/" are positional-only parameters and may only be passed by positional arguments.

在 3.8 版的變更: The / function parameter syntax may be used to indicate positional-only parameters. See **PEP 570** for details.

Parameters may have an *annotation* of the form "`: expression`" following the parameter name. Any parameter may have an annotation, even those of the form `*identifier` or `**identifier`. (As a special case, parameters of the form `*identifier` may have an annotation "`: *expression`".) Functions may have "return" annotation of the form "`-> expression`" after the parameter list. These annotations can be any valid Python expression. The presence of annotations does not change the semantics of a function. The annotation values are available as values of a dictionary keyed by the parameters' names in the `__annotations__` attribute of the function object. If the `annotations` import from `__future__` is used, annotations are preserved as strings at runtime which enables postponed evaluation. Otherwise, they are evaluated when the function definition is executed. In this case annotations may be evaluated in a different order than they appear in the source code.

在 3.11 版的變更: Parameters of the form "`*identifier`" may have an annotation "`: *expression`". See **PEP 646**.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda expressions, described in section *Lambdas*. Note that the lambda expression is merely a shorthand for a simplified function definition; a function defined in a "`def`" statement can be passed around or assigned to another name just like a function defined by a lambda expression. The "`def`" form is actually more powerful since it allows the execution of multiple statements and annotations.

Programmer's note: Functions are first-class objects. A "`def`" statement executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section *Naming and binding* for details.

> ➴ **也參考**
>
> **PEP 3107 - Function Annotations**
>   The original specification for function annotations.
>
> **PEP 484 - Type Hints**
>   Definition of a standard meaning for annotations: type hints.
>
> **PEP 526 - Syntax for Variable Annotations**
>   Ability to type hint variable declarations, including class variables and instance variables.

> **PEP 563 - Postponed Evaluation of Annotations**
> Support for forward references within annotations by preserving annotations in a string form at runtime instead of eager evaluation.
>
> **PEP 318 - Decorators for Functions and Methods**
> Function and method decorators were introduced. Class decorators were introduced in **PEP 3129**.

8.8 類⬚定義

A class definition defines a class object (see section 標準型⬚階層):

```
classdef    ::= [decorators] "class" classname [type_params] [inheritance] ":" suite
inheritance ::= "(" [argument_list] ")"
classname   ::= identifier
```

A class definition is an executable statement. The inheritance list usually gives a list of base classes (see *Metaclasses* for more advanced uses), so each item in the list should evaluate to a class object which allows subclassing. Classes without an inheritance list inherit, by default, from the base class `object`; hence,

```
class Foo:
    pass
```

is equivalent to

```
class Foo(object):
    pass
```

The class's suite is then executed in a new execution frame (see *Naming and binding*), using a newly created local namespace and the original global namespace. (Usually, the suite contains mostly function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved.[5] A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

The order in which attributes are defined in the class body is preserved in the new class's `__dict__`. Note that this is reliable only right after the class is created and only for classes that were defined using the definition syntax.

Class creation can be customized heavily using *metaclasses*.

Classes can also be decorated: just like when decorating functions,

```
@f1(arg)
@f2
class Foo: pass
```

大致等價於:

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

The evaluation rules for the decorator expressions are the same as for function decorators. The result is then bound to the class name.

在 3.9 版的變更: Classes may be decorated with any valid `assignment_expression`. Previously, the grammar was much more restrictive; see **PEP 614** for details.

A list of *type parameters* may be given in square brackets immediately after the class's name. This indicates to static type checkers that the class is generic. At runtime, the type parameters can be retrieved from the class's `__type_params__` attribute. See *Generic classes* for more.

---

[5] A string literal appearing as the first statement in the class body is transformed into the namespace's `__doc__` item and therefore the class's *docstring*.

在 3.12 版的變更: Type parameter lists are new in Python 3.12.

Programmer's note: Variables defined in the class definition are class attributes; they are shared by instances. Instance attributes can be set in a method with `self.name = value`. Both class and instance attributes are accessible through the notation "self.name", and an instance attribute hides a class attribute with the same name when accessed in this way. Class attributes can be used as defaults for instance attributes, but using mutable values there can lead to unexpected results. *Descriptors* can be used to create instance variables with different implementation details.

> ↪ **也參考**
>
> **PEP 3115 - Metaclasses in Python 3000**
>   The proposal that changed the declaration of metaclasses to the current syntax, and the semantics for how classes with metaclasses are constructed.
>
> **PEP 3129 - 類⊞裝飾器**
>   The proposal that added class decorators. Function and method decorators were introduced in **PEP 318**.

8.9 協程

在 3.5 版被加入.

8.9.1 協程函式定義

```
async_funcdef ::= [decorators] "async" "def" funcname "(" [parameter_list] ")"
                  ["->" expression] ":" suite
```

Execution of Python coroutines can be suspended and resumed at many points (see *coroutine*). *await* expressions, *async for* and *async with* can only be used in the body of a coroutine function.

Functions defined with `async def` syntax are always coroutine functions, even if they do not contain `await` or `async` keywords.

It is a `SyntaxError` to use a `yield from` expression inside the body of a coroutine function.

一個協程函式範例:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

在 3.7 版的變更: `await` and `async` are now keywords; previously they were only treated as such inside the body of a coroutine function.

8.9.2 `async for` 陳述式

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

An *asynchronous iterable* provides an `__aiter__` method that directly returns an *asynchronous iterator*, which can call asynchronous code in its `__anext__` method.

The `async for` statement allows convenient iteration over asynchronous iterables.

以下程式碼:

```
async for TARGET in ITER:
    SUITE
else:
    SUITE2
```

Is semantically equivalent to:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True

while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        SUITE
else:
    SUITE2
```

更多細節請見`__aiter__()`與`__anext__()`。

It is a SyntaxError to use an `async for` statement outside the body of a coroutine function.

8.9.3 `async with` 陳述式

```
async_with_stmt ::= "async" with_stmt
```

An *asynchronous context manager* is a *context manager* that is able to suspend execution in its *enter* and *exit* methods.

以下程式碼：

```
async with EXPRESSION as TARGET:
    SUITE
```

在語義上等同於：

```
manager = (EXPRESSION)
aenter = type(manager).__aenter__
aexit = type(manager).__aexit__
value = await aenter(manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not await aexit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit(manager, None, None, None)
```

更多細節請見`__aenter__()`與`__aexit__()`。

It is a SyntaxError to use an `async with` statement outside the body of a coroutine function.

> **↪ 也參考**
>
> **PEP 492** - **Coroutines with async and await syntax**
> The proposal that made coroutines a proper standalone concept in Python, and added supporting syntax.

8.10 Type parameter lists

在 3.12 版被加入.

在 3.13 版的變更: Support for default values was added (see **PEP 696**).

```
type_params   ::= "[" type_param ("," type_param)* "]"
type_param    ::= typevar | typevartuple | paramspec
typevar       ::= identifier (":" expression)? ("=" expression)?
typevartuple  ::= "*" identifier ("=" expression)?
paramspec     ::= "**" identifier ("=" expression)?
```

Functions (including *coroutines*), *classes* and *type aliases* may contain a type parameter list:

```python
def max[T](args: list[T]) -> T:
    ...

async def amax[T](args: list[T]) -> T:
    ...

class Bag[T]:
    def __iter__(self) -> Iterator[T]:
        ...

    def add(self, arg: T) -> None:
        ...

type ListOrSet[T] = list[T] | set[T]
```

Semantically, this indicates that the function, class, or type alias is generic over a type variable. This information is primarily used by static type checkers, and at runtime, generic objects behave much like their non-generic counterparts.

Type parameters are declared in square brackets (`[]`) immediately after the name of the function, class, or type alias. The type parameters are accessible within the scope of the generic object, but not elsewhere. Thus, after a declaration `def func[T](): pass`, the name `T` is not available in the module scope. Below, the semantics of generic objects are described with more precision. The scope of type parameters is modeled with a special function (technically, an *annotation scope*) that wraps the creation of the generic object.

Generic functions, classes, and type aliases have a `__type_params__` attribute listing their type parameters.

Type parameters come in three kinds:

- `typing.TypeVar`, introduced by a plain name (e.g., `T`). Semantically, this represents a single type to a type checker.

- `typing.TypeVarTuple`, introduced by a name prefixed with a single asterisk (e.g., `*Ts`). Semantically, this stands for a tuple of any number of types.

- `typing.ParamSpec`, introduced by a name prefixed with two asterisks (e.g., `**P`). Semantically, this stands for the parameters of a callable.

`typing.TypeVar` declarations can define *bounds* and *constraints* with a colon (`:`) followed by an expression. A single expression after the colon indicates a bound (e.g. `T: int`). Semantically, this means that the `typing.TypeVar` can only represent types that are a subtype of this bound. A parenthesized tuple of expressions after the colon indicates a set of constraints (e.g. `T: (str, bytes)`). Each member of the tuple should be a type (again, this is not enforced at runtime). Constrained type variables can only take on one of the types in the list of constraints.

For `typing.TypeVar`s declared using the type parameter list syntax, the bound and constraints are not evaluated when the generic object is created, but only when the value is explicitly accessed through the attributes `__bound__` and `__constraints__`. To accomplish this, the bounds or constraints are evaluated in a separate *annotation scope*.

`typing.TypeVarTuple`s and `typing.ParamSpec`s cannot have bounds or constraints.

All three flavors of type parameters can also have a *default value*, which is used when the type parameter is not explicitly provided. This is added by appending a single equals sign (=) followed by an expression. Like the bounds and constraints of type variables, the default value is not evaluated when the object is created, but only when the type parameter's `__default__` attribute is accessed. To this end, the default value is evaluated in a separate *annotation scope*. If no default value is specified for a type parameter, the `__default__` attribute is set to the special sentinel object `typing.NoDefault`.

The following example indicates the full set of allowed type parameter declarations:

```python
def overly_generic[
    SimpleTypeVar,
    TypeVarWithDefault = int,
    TypeVarWithBound: int,
    TypeVarWithConstraints: (str, bytes),
    *SimpleTypeVarTuple = (int, float),
    **SimpleParamSpec = (str, bytearray),
](
    a: SimpleTypeVar,
    b: TypeVarWithDefault,
    c: TypeVarWithBound,
    d: Callable[SimpleParamSpec, TypeVarWithConstraints],
    *e: SimpleTypeVarTuple,
): ...
```

8.10.1 Generic functions

Generic functions are declared as follows:

```python
def func[T](arg: T): ...
```

語法大致等價於：

```python
annotation-def TYPE_PARAMS_OF_func():
    T = typing.TypeVar("T")
    def func(arg: T): ...
    func.__type_params__ = (T,)
    return func
func = TYPE_PARAMS_OF_func()
```

Here `annotation-def` indicates an *annotation scope*, which is not actually bound to any name at runtime. (One other liberty is taken in the translation: the syntax does not go through attribute access on the `typing` module, but creates an instance of `typing.TypeVar` directly.)

The annotations of generic functions are evaluated within the annotation scope used for declaring the type parameters, but the function's defaults and decorators are not.

The following example illustrates the scoping rules for these cases, as well as for additional flavors of type parameters:

```python
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = some_default):
    ...
```

Except for the *lazy evaluation* of the `TypeVar` bound, this is equivalent to:

```python
DEFAULT_OF_arg = some_default

annotation-def TYPE_PARAMS_OF_func():

    annotation-def BOUND_OF_T():
        return int
    # In reality, BOUND_OF_T() is evaluated only on demand.
    T = typing.TypeVar("T", bound=BOUND_OF_T())
```

```
    Ts = typing.TypeVarTuple("Ts")
    P = typing.ParamSpec("P")

    def func(*args: *Ts, arg: Callable[P, T] = DEFAULT_OF_arg):
        ...

    func.__type_params__ = (T, Ts, P)
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

The capitalized names like DEFAULT_OF_arg are not actually bound at runtime.

8.10.2 Generic classes

Generic classes are declared as follows:

```
class Bag[T]: ...
```

語法大致等價於：

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(typing.Generic[T]):
        __type_params__ = (T,)
        ...
    return Bag
Bag = TYPE_PARAMS_OF_Bag()
```

Here again annotation-def (not a real keyword) indicates an *annotation scope*, and the name TYPE_PARAMS_OF_Bag is not actually bound at runtime.

Generic classes implicitly inherit from typing.Generic. The base classes and keyword arguments of generic classes are evaluated within the type scope for the type parameters, and decorators are evaluated outside that scope. This is illustrated by this example:

```
@decorator
class Bag(Base[T], arg=T): ...
```

這等價於：

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(Base[T], typing.Generic[T], arg=T):
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

8.10.3 Generic type aliases

The *type* statement can also be used to create a generic type alias:

```
type ListOrSet[T] = list[T] | set[T]
```

Except for the *lazy evaluation* of the value, this is equivalent to:

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")
```

```
    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
    # In reality, the value is lazily evaluated
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_params=(T,))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

Here, `annotation-def` (not a real keyword) indicates an *annotation scope*. The capitalized names like `TYPE_PARAMS_OF_ListOrSet` are not actually bound at runtime.

F**解**

最高層級元件

Python 直譯器可以從多種來源獲得輸入：作為標准輸入或程式引數自腳本傳入、以互動式鍵入、自模組原始檔引入等等。這一章將給出在這些情況下所用的語法。

9.1 完整的 Python 程式

雖然語言規範描述不必規定語言直譯器是如何被調用（invoke）的，但對完整的 Python 程式加以說明還是很有用的。一個完整的 Python 程式會在最小初始化環境中被執行：所有內建和標准模組均為可用，但均處於未初始化狀態，只有 sys（各種系統服務）、builtins（內建函式、例外和 None）和 __main__ 除外。後者用於為完整程式的執行提供區域性和全域性命名空間 (namespace)。

用於一個完整 Python 程式的語法，即下節所描述的檔案輸入。

直譯器也可以透過互動模式被調用；在此情況下，它並不讀取和執行一個完整程式，而是每讀取一條陳述式就執行一次（可能是複合陳述式）。此時的初始環境與一個完整程式的相同；每條陳述式會在 __main__ 的命名空間中被執行。

一個完整程式可透過三種形式被傳遞給直譯器：使用 -c 字串命令列選項、使用一個檔案作為第一個命令列引數、或者使用標准輸入。如果檔案或標准輸入是一個 tty 裝置，直譯器會進入互動模式；否則它會將檔案當作一個完整程式來執行。

9.2 檔案輸入

所有從非互動式檔案讀取的輸入都具有相同的形式：

```
file_input ::= (NEWLINE | statement)*
```

此語法用於下列幾種情況：

- 剖析一個完整 Python 程式時（從檔案或字串）；
- 剖析一個模組時；
- 剖析一個傳遞給 exec() 函式的字串時；

9.3 互動式輸入

互動模式下的輸入使用以下語法進行剖析：

```
interactive_input ::= [stmt_list] NEWLINE | compound_stmt NEWLINE
```

注意在互動模式下，一條（最高層級）⊡合陳述式最後必須帶有一個空行；這能⊡幫助剖析器確定輸入已經結束。

9.4 運算式輸入

eval() 被用於運算式輸入，它會忽略開頭的空白。傳遞給 eval() 的字串引數必須具有以下形式：

```
eval_input ::= expression_list NEWLINE*
```

CHAPTER 10

CHAPTER 10

完整的語法規格書

This is the full Python grammar, derived directly from the grammar used to generate the CPython parser (see Grammar/python.gram). The version here omits details related to code generation and error recovery.

The notation is a mixture of EBNF and PEG. In particular, & followed by a symbol, token or parenthesized group indicates a positive lookahead (i.e., is required to match but not consumed), while ! indicates a negative lookahead (i.e., is required *not* to match). We use the | separator to mean PEG's "ordered choice" (written as / in traditional PEG grammars). See **PEP 617** for more details on the grammar's syntax.

```
# PEG grammar for Python



# ========================= START OF THE GRAMMAR =========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic syntax error, the
#       location of the generic failure of the first pass will be used (this avoids
#       reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#       (like any rule in PEG).
#
# Grammar Syntax (see PEP 617 for more information):
#
# rule_name: expression
#   Optionally, a type can be included right after the rule name, which
#   specifies the return type of the C or Python function corresponding to the
#   rule:
# rule_name[return_type]: expression
#   If the return type is omitted, then a void * is returned in C and an Any in
#   Python.
# e1 e2
```

```
#   Match e1, then match e2.
# e1 | e2
#   Match e1 or e2.
#   The first alternative can also appear on the line after the rule name for
#   formatting purposes. In that case, a | must be used before the first
#   alternative, like so:
#       rule_name[return_type]:
#           | first_alt
#           | second_alt
# ( e )
#   Match e (allows also to use other operators in the group like '(e)*')
# [ e ] or e?
#   Optionally match e.
# e*
#   Match zero or more occurrences of e.
# e+
#   Match one or more occurrences of e.
# s.e+
#   Match one or more occurrences of e, separated by s. The generated parse tree
#   does not include the separator. This is otherwise identical to (e (s e)*).
# &e
#   Succeed if e can be parsed, without consuming any input.
# !e
#   Fail if e can be parsed, without consuming any input.
# ~
#   Commit to the current alternative, even if it fails to parse.
# &&e
#   Eager parse e. The parser will not backtrack and will immediately
#   fail with SyntaxError if e cannot be parsed.
#

# STARTING RULES
# ==============

file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER

# GENERAL STATEMENTS
# ==================

statements: statement+

statement: compound_stmt  | simple_stmts

statement_newline:
    | compound_stmt NEWLINE
    | simple_stmts
    | NEWLINE
    | ENDMARKER

simple_stmts:
    | simple_stmt !';' NEWLINE  # Not needed, there for speedup
    | ';'.simple_stmt+ [';'] NEWLINE

# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt:
    | assignment
    | type_alias
```

```
    | star_expressions
    | return_stmt
    | import_stmt
    | raise_stmt
    | 'pass'
    | del_stmt
    | yield_stmt
    | assert_stmt
    | 'break'
    | 'continue'
    | global_stmt
    | nonlocal_stmt

compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | with_stmt
    | for_stmt
    | try_stmt
    | while_stmt
    | match_stmt

# SIMPLE STATEMENTS
# =================

# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with 'yield'
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | ('(' single_target ')'
         | single_subscript_attribute_target) ':' expression ['=' annotated_rhs ]
    | (star_targets '=' )+ (yield_expr | star_expressions) !'=' [TYPE_COMMENT]
    | single_target augassign ~ (yield_expr | star_expressions)

annotated_rhs: yield_expr | star_expressions

augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

global_stmt: 'global' ','.NAME+

nonlocal_stmt: 'nonlocal' ','.NAME+
```

```
del_stmt:
    | 'del' del_targets &(';' | NEWLINE)

yield_stmt: yield_expr

assert_stmt: 'assert' expression [',' expression ]

import_stmt:
    | import_name
    | import_from

# Import statements
# -----------------

import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from:
    | 'from' ('.' | '...')* dotted_name 'import' import_from_targets
    | 'from' ('.' | '...')+ 'import' import_from_targets
import_from_targets:
    | '(' import_from_as_names [','] ')'
    | import_from_as_names !','
    | '*'
import_from_as_names:
    | ','.import_from_as_name+
import_from_as_name:
    | NAME ['as' NAME ]
dotted_as_names:
    | ','.dotted_as_name+
dotted_as_name:
    | dotted_name ['as' NAME ]
dotted_name:
    | dotted_name '.' NAME
    | NAME

# COMPOUND STATEMENTS
# ===================

# Common elements
# ---------------

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmts

decorators: ('@' named_expression NEWLINE )+

# Class definitions
# -----------------

class_def:
    | decorators class_def_raw
    | class_def_raw

class_def_raw:
    | 'class' NAME [type_params] ['(' [arguments] ')' ] ':' block

# Function definitions
# --------------------
```

```
function_def:
    | decorators function_def_raw
    | function_def_raw

function_def_raw:
    | 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_type_comment]␣
↪block
    | 'async' 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_type_
↪comment] block

# Function parameters
# -------------------

params:
    | parameters

parameters:
    | slash_no_default param_no_default* param_with_default* [star_etc]
    | slash_with_default param_with_default* [star_etc]
    | param_no_default+ param_with_default* [star_etc]
    | param_with_default+ [star_etc]
    | star_etc

# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).

slash_no_default:
    | param_no_default+ '/' ','
    | param_no_default+ '/' &')'
slash_with_default:
    | param_no_default* param_with_default+ '/' ','
    | param_no_default* param_with_default+ '/' &')'

star_etc:
    | '*' param_no_default param_maybe_default* [kwds]
    | '*' param_no_default_star_annotation param_maybe_default* [kwds]
    | '*' ',' param_maybe_default+ [kwds]
    | kwds

kwds:
    | '**' param_no_default

# One parameter.  This *includes* a following comma and type comment.
#
# There are three styles:
# - No default
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
#

param_no_default:
    | param ',' TYPE_COMMENT?
    | param TYPE_COMMENT? &')'
param_no_default_star_annotation:
    | param_star_annotation ',' TYPE_COMMENT?
    | param_star_annotation TYPE_COMMENT? &')'
```

```
param_with_default:
    | param default ',' TYPE_COMMENT?
    | param default TYPE_COMMENT? &')'
param_maybe_default:
    | param default? ',' TYPE_COMMENT?
    | param default? TYPE_COMMENT? &')'
param: NAME annotation?
param_star_annotation: NAME star_annotation
annotation: ':' expression
star_annotation: ':' star_expression
default: '=' expression  | invalid_default

# If statement
# ------------

if_stmt:
    | 'if' named_expression ':' block elif_stmt
    | 'if' named_expression ':' block [else_block]
elif_stmt:
    | 'elif' named_expression ':' block elif_stmt
    | 'elif' named_expression ':' block [else_block]
else_block:
    | 'else' ':' block

# While statement
# ---------------

while_stmt:
    | 'while' named_expression ':' block [else_block]

# For statement
# -------------

for_stmt:
    | 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_block]
    | 'async' 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_
↪block]

# With statement
# --------------

with_stmt:
    | 'with' '(' ','.with_item+ ','? ')' ':' [TYPE_COMMENT] block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | 'async' 'with' '(' ','.with_item+ ','? ')' ':' block
    | 'async' 'with' ','.with_item+ ':' [TYPE_COMMENT] block

with_item:
    | expression 'as' star_target &(',' | ')' | ':')
    | expression

# Try statement
# -------------

try_stmt:
    | 'try' ':' block finally_block
    | 'try' ':' block except_block+ [else_block] [finally_block]
    | 'try' ':' block except_star_block+ [else_block] [finally_block]


# Except statement
```

```
# ----------------

except_block:
    | 'except' expression ['as' NAME ] ':' block
    | 'except' ':' block
except_star_block:
    | 'except' '*' expression ['as' NAME ] ':' block
finally_block:
    | 'finally' ':' block

# Match statement
# --------------

match_stmt:
    | "match" subject_expr ':' NEWLINE INDENT case_block+ DEDENT

subject_expr:
    | star_named_expression ',' star_named_expressions?
    | named_expression

case_block:
    | "case" patterns guard? ':' block

guard: 'if' named_expression

patterns:
    | open_sequence_pattern
    | pattern

pattern:
    | as_pattern
    | or_pattern

as_pattern:
    | or_pattern 'as' pattern_capture_target

or_pattern:
    | '|'.closed_pattern+

closed_pattern:
    | literal_pattern
    | capture_pattern
    | wildcard_pattern
    | value_pattern
    | group_pattern
    | sequence_pattern
    | mapping_pattern
    | class_pattern

# Literal patterns are used for equality and identity constraints
literal_pattern:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

# Literal expressions are used to restrict permitted mapping pattern keys
literal_expr:
    | signed_number !('+' | '-')
```

```
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

complex_number:
    | signed_real_number '+' imaginary_number
    | signed_real_number '-' imaginary_number

signed_number:
    | NUMBER
    | '-' NUMBER

signed_real_number:
    | real_number
    | '-' real_number

real_number:
    | NUMBER

imaginary_number:
    | NUMBER

capture_pattern:
    | pattern_capture_target

pattern_capture_target:
    | !"_" NAME !('.' | '(' | '=')

wildcard_pattern:
    | "_"

value_pattern:
    | attr !('.' | '(' | '=')

attr:
    | name_or_attr '.' NAME

name_or_attr:
    | attr
    | NAME

group_pattern:
    | '(' pattern ')'

sequence_pattern:
    | '[' maybe_sequence_pattern? ']'
    | '(' open_sequence_pattern? ')'

open_sequence_pattern:
    | maybe_star_pattern ',' maybe_sequence_pattern?

maybe_sequence_pattern:
    | ','.maybe_star_pattern+ ','?

maybe_star_pattern:
    | star_pattern
    | pattern

star_pattern:
```

```
      | '*' pattern_capture_target
      | '*' wildcard_pattern

mapping_pattern:
      | '{' '}'
      | '{' double_star_pattern ','? '}'
      | '{' items_pattern ',' double_star_pattern ','? '}'
      | '{' items_pattern ','? '}'

items_pattern:
      | ','.key_value_pattern+

key_value_pattern:
      | (literal_expr | attr) ':' pattern

double_star_pattern:
      | '**' pattern_capture_target

class_pattern:
      | name_or_attr '(' ')'
      | name_or_attr '(' positional_patterns ','? ')'
      | name_or_attr '(' keyword_patterns ','? ')'
      | name_or_attr '(' positional_patterns ',' keyword_patterns ','? ')'

positional_patterns:
      | ','.pattern+

keyword_patterns:
      | ','.keyword_pattern+

keyword_pattern:
      | NAME '=' pattern

# Type statement
# --------------

type_alias:
      | "type" NAME [type_params] '=' expression

# Type parameter declaration
# --------------------------

type_params:
      | invalid_type_params
      | '[' type_param_seq ']'

type_param_seq: ','.type_param+ [',']

type_param:
      | NAME [type_param_bound] [type_param_default]
      | '*' NAME [type_param_starred_default]
      | '**' NAME [type_param_default]

type_param_bound: ':' expression
type_param_default: '=' expression
type_param_starred_default: '=' star_expression

# EXPRESSIONS
# -----------

expressions:
```

```
    | expression (',' expression )+ [',']
    | expression ','
    | expression

expression:
    | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef

yield_expr:
    | 'yield' 'from' expression
    | 'yield' [star_expressions]

star_expressions:
    | star_expression (',' star_expression )+ [',']
    | star_expression ','
    | star_expression

star_expression:
    | '*' bitwise_or
    | expression

star_named_expressions: ','.star_named_expression+ [',']

star_named_expression:
    | '*' bitwise_or
    | named_expression

assignment_expression:
    | NAME ':=' ~ expression

named_expression:
    | assignment_expression
    | expression !':='

disjunction:
    | conjunction ('or' conjunction )+
    | conjunction

conjunction:
    | inversion ('and' inversion )+
    | inversion

inversion:
    | 'not' inversion
    | comparison

# Comparison operators
# --------------------

comparison:
    | bitwise_or compare_op_bitwise_or_pair+
    | bitwise_or

compare_op_bitwise_or_pair:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
```

```
    | notin_bitwise_or
    | in_bitwise_or
    | isnot_bitwise_or
    | is_bitwise_or

eq_bitwise_or: '==' bitwise_or
noteq_bitwise_or:
    | ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or
lt_bitwise_or: '<' bitwise_or
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or

# Bitwise operators
# -----------------

bitwise_or:
    | bitwise_or '|' bitwise_xor
    | bitwise_xor

bitwise_xor:
    | bitwise_xor '^' bitwise_and
    | bitwise_and

bitwise_and:
    | bitwise_and '&' shift_expr
    | shift_expr

shift_expr:
    | shift_expr '<<' sum
    | shift_expr '>>' sum
    | sum

# Arithmetic operators
# --------------------

sum:
    | sum '+' term
    | sum '-' term
    | term

term:
    | term '*' factor
    | term '/' factor
    | term '//' factor
    | term '%' factor
    | term '@' factor
    | factor

factor:
    | '+' factor
    | '-' factor
    | '~' factor
    | power

power:
    | await_primary '**' factor
```

```
    | await_primary

# Primary elements
# ----------------

# Primary elements are things like "obj.something.something", "obj[something]",
→"obj(something)", "obj" ...

await_primary:
    | 'await' primary
    | primary

primary:
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice !','
    | ','.(slice | starred_expression)+ [',']

slice:
    | [expression] ':' [expression] [':' [expression] ]
    | named_expression

atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

group:
    | '(' (yield_expr | named_expression) ')'

# Lambda functions
# ----------------

lambdef:
    | 'lambda' [lambda_params] ':' expression

lambda_params:
    | lambda_parameters

# lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis.  (For more, see parameters above.)
#
lambda_parameters:
    | lambda_slash_no_default lambda_param_no_default* lambda_param_with_default* [lambda_
→star_etc]
    | lambda_slash_with_default lambda_param_with_default* [lambda_star_etc]
    | lambda_param_no_default+ lambda_param_with_default* [lambda_star_etc]
    | lambda_param_with_default+ [lambda_star_etc]
```

```
    | lambda_star_etc

lambda_slash_no_default:
    | lambda_param_no_default+ '/' ','
    | lambda_param_no_default+ '/' &':'

lambda_slash_with_default:
    | lambda_param_no_default* lambda_param_with_default+ '/' ','
    | lambda_param_no_default* lambda_param_with_default+ '/' &':'

lambda_star_etc:
    | '*' lambda_param_no_default lambda_param_maybe_default* [lambda_kwds]
    | '*' ',' lambda_param_maybe_default+ [lambda_kwds]
    | lambda_kwds

lambda_kwds:
    | '**' lambda_param_no_default

lambda_param_no_default:
    | lambda_param ','
    | lambda_param &':'
lambda_param_with_default:
    | lambda_param default ','
    | lambda_param default &':'
lambda_param_maybe_default:
    | lambda_param default? ','
    | lambda_param default? &':'
lambda_param: NAME

# LITERALS
# ========

fstring_middle:
    | fstring_replacement_field
    | FSTRING_MIDDLE
fstring_replacement_field:
    | '{' annotated_rhs '='? [fstring_conversion] [fstring_full_format_spec] '}'
fstring_conversion:
    | "!" NAME
fstring_full_format_spec:
    | ':' fstring_format_spec*
fstring_format_spec:
    | FSTRING_MIDDLE
    | fstring_replacement_field
fstring:
    | FSTRING_START fstring_middle* FSTRING_END

string: STRING
strings: (fstring|string)+

list:
    | '[' [star_named_expressions] ']'

tuple:
    | '(' [star_named_expression ',' [star_named_expressions]  ] ')'

set: '{' star_named_expressions '}'

# Dicts
# -----
```

```
dict:
    | '{' [double_starred_kvpairs] '}'

double_starred_kvpairs: ','.double_starred_kvpair+ [',']

double_starred_kvpair:
    | '**' bitwise_or
    | kvpair

kvpair: expression ':' expression

# Comprehensions & Generators
# --------------------------

for_if_clauses:
    | for_if_clause+

for_if_clause:
    | 'async' 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
    | 'for' star_targets 'in' ~ disjunction ('if' disjunction )*

listcomp:
    | '[' named_expression for_if_clauses ']'

setcomp:
    | '{' named_expression for_if_clauses '}'

genexp:
    | '(' ( assignment_expression | expression !':=') for_if_clauses ')'

dictcomp:
    | '{' kvpair for_if_clauses '}'

# FUNCTION CALL ARGUMENTS
# =======================

arguments:
    | args [','] &')'

args:
    | ','.(starred_expression | ( assignment_expression | expression !':=') !'=')+ [','␣
↪kwargs ]
    | kwargs

kwargs:
    | ','.kwarg_or_starred+ ',' ','.kwarg_or_double_starred+
    | ','.kwarg_or_starred+
    | ','.kwarg_or_double_starred+

starred_expression:
    | '*' expression

kwarg_or_starred:
    | NAME '=' expression
    | starred_expression

kwarg_or_double_starred:
    | NAME '=' expression
    | '**' expression

# ASSIGNMENT TARGETS
```

```
# ==================

# Generic targets
# ---------------

# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
    | star_target !','
    | star_target (',' star_target )* [',']

star_targets_list_seq: ','.star_target+ [',']

star_targets_tuple_seq:
    | star_target (',' star_target )+ [',']
    | star_target ','

star_target:
    | '*' (!'*' star_target)
    | target_with_star_atom

target_with_star_atom:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom

star_atom:
    | NAME
    | '(' target_with_star_atom ')'
    | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'

single_target:
    | single_subscript_attribute_target
    | NAME
    | '(' single_target ')'

single_subscript_attribute_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead

t_primary:
    | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
    | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead
    | atom &t_lookahead

t_lookahead: '(' | '[' | '.'

# Targets for del statements
# --------------------------

del_targets: ','.del_target+ [',']

del_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | del_t_atom

del_t_atom:
    | NAME
```

```
    | '(' del_target ')'
    | '(' [del_targets] ')'
    | '[' [del_targets] ']'

# TYPING ELEMENTS
# ---------------

# type_expressions allow */** but ignore them
type_expressions:
    | ','.expression+ ',' '*' expression ',' '**' expression
    | ','.expression+ ',' '*' expression
    | ','.expression+ ',' '**' expression
    | '*' expression ',' '**' expression
    | '*' expression
    | '**' expression
    | ','.expression+

func_type_comment:
    | NEWLINE TYPE_COMMENT &(NEWLINE INDENT)    # Must be followed by indented block
    | TYPE_COMMENT

# ========================= END OF THE GRAMMAR =========================



# ========================= START OF INVALID RULES =========================
```

術語表

>>>
　　互動式 shell 的預設 Python 提示字元。常見於能在直譯器中以互動方式被執行的程式碼範例。

...
　　可以表示：

- 在一個被縮排的程式碼區塊、在一對匹配的左右定界符（delimiter，例如括號、方括號、花括號或三引號）內部，或是在指定一個裝飾器 (decorator) 之後，要輸入程式碼時，互動式 shell 顯示的預設 Python 提示字元。
- 內建常數 Ellipsis。

abstract base class（抽象基底類別）
　　抽象基底類別（又稱為 ABC）提供了一種定義介面的方法，作為 *duck-typing*（鴨子型別）的補充。其他類似的技術，像是 hasattr()，則顯得笨拙或是帶有細微的錯誤（例如使用*魔術方法 (magic method)*）。ABC 引入虛擬的 subclass（子類別），它們並不繼承自另一個 class（類別），但仍可被 isinstance() 及 issubclass() 辨識；請參閱 abc 模組的說明文件。Python 有許多內建的 ABC，用於資料結構（在 collections.abc 模組）、數字（在 numbers 模組）、串流（在 io 模組）及 import 尋檢器和載入器（在 importlib.abc 模組）。你可以使用 abc 模組建立自己的 ABC。

annotation（註釋）
　　一個與變數、class 屬性、函式的參數或回傳值相關聯的標籤。照慣例，它被用來作為 *type hint*（型別提示）。

　　在執行環境 (runtime)，區域變數的註釋無法被存取，但全域變數、class 屬性和函式的註解，會分別被儲存在模組、class 和函式的 __annotations__ 特殊屬性中。

　　請參閱 *variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，這些章節皆有此功能的說明。關於註釋的最佳實踐方法也請參閱 annotations-howto。

argument（引數）
　　呼叫函式時被傳遞給 *function*（或 *method*）的值。引數有兩種：

- 關鍵字引數 *(keyword argument)*：在函式呼叫中，以識別字（identifier，例如 name=）開頭的引數，或是以 ** 後面 dictionary（字典）中的值被傳遞的引數。例如，3 和 5 都是以下 complex() 呼叫中的關鍵字引數：

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- 位置引數 (*positional argument*)：不是關鍵字引數的引數。位置引數可在一個引數列表的起始處出現，和（或）作⿰ `*` 之後的*iterable*（可⿰代物件）中的元素被傳遞。例如，3 和 5 都是以下呼叫中的位置引數：

```
complex(3, 5)
complex(*(3, 5))
```

引數會被指定給函式主體中的附名區域變數。關於支配這個指定過程的規則，請參⿰*Calls*章節。在語法上，任何運算式都可以被用來表示一個引數；其評估值會被指定給區域變數。

另請參⿰術語表的*parameter*（參數）條目、常見問題中的引數和參數之間的差⿰，以及 **PEP 362**。

asynchronous context manager（非同步情境管理器）

一個可以控制`async with`陳述式中所見環境的物件，而它是透過定義`__aenter__()`和`__aexit__()` method（方法）來控制的。由 **PEP 492** 引入。

asynchronous generator（非同步⿰生器）

一個會回傳*asynchronous generator iterator*（非同步⿰生器⿰代器）的函式。它看起來像一個以`async def` 定義的協程函式 (coroutine function)，但不同的是它包含了`yield` 運算式，能生成一系列可用於`async for`⿰圈的值。

這個術語通常用來表示一個非同步⿰生器函式，但在某些情境中，也可能是表示非同步⿰生器⿰代器 (*asynchronous generator iterator*)。萬一想表達的意思不⿰清楚，那就使用完整的術語，以避免歧義。

一個非同步⿰生器函式可能包含`await` 運算式，以及`async for`和`async with` 陳述式。

asynchronous generator iterator（非同步⿰生器⿰代器）

一個由*asynchronous generator*（非同步⿰生器）函式所建立的物件。

這是一個*asynchronous iterator*（非同步⿰代器），當它以`__anext__()` method 被呼叫時，會回傳一個可等待物件 (awaitable object)，該物件將執行非同步⿰生器的函式主體，直到遇到下一個`yield` 運算式。

每個`yield`會暫停處理程序，⿰記住執行狀態（包括區域變數及擱置中的 try 陳述式）。當非同步⿰生器⿰代器以另一個被`__anext__()`回傳的可等待物件有效地回復時，它會從停止的地方繼續執行。請參⿰ **PEP 492** 和 **PEP 525**。

asynchronous iterable（非同步可⿰代物件）

一個物件，它可以在`async for` 陳述式中被使用。必須從它的`__aiter__()` method 回傳一個*asynchronous iterator*（非同步⿰代器）。由 **PEP 492** 引入。

asynchronous iterator（非同步⿰代器）

一個實作`__aiter__()` 和`__anext__()` method 的物件。`__anext__()` 必須回傳一個*awaitable*（可等待物件）。`async for` 會解析非同步⿰代器的`__anext__()` method 所回傳的可等待物件，直到它引發 StopAsyncIteration 例外。由 **PEP 492** 引入。

attribute（屬性）

一個與某物件相關聯的值，該值大多能透過使用點分隔運算式 (dotted expression) 的名稱被參照。例如，如果物件 *o* 有一個屬性 *a*，則該屬性能以 *o.a* 被參照。

如果一個物件允許，給予該物件一個名稱不是由*Identifiers and keywords*所定義之識⿰符 (identifier) 的屬性是有可能的，例如使用 setattr()。像這樣的屬性將無法使用點分隔運算式來存取，而是需要使用 getattr() 來取得它。

awaitable（可等待物件）

一個可以在`await` 運算式中被使用的物件。它可以是一個*coroutine*（協程），或是一個有`__await__()` method 的物件。另請參⿰ **PEP 492**。

BDFL

Benevolent Dictator For Life（終身仁慈獨裁者），又名 Guido van Rossum，Python 的創造者。

binary file（二進位檔案）

一個能⿰讀取和寫入*bytes-like objects*（類位元組串物件）的*file object*（檔案物件）。二進位檔案的例子有：以二進位模式（'rb'、'wb' 或 'rb+'）開⿰的檔案、sys.stdin.buffer、sys.stdout.buffer，以及 io.BytesIO 和 gzip.GzipFile 實例。

另請參見 *text file*（文字檔案），它是一個能夠讀取和寫入 `str` 物件的檔案物件。

borrowed reference（借用參照）

在 Python 的 C API 中，借用參照是一個對物件的參照，其中使用該物件的程式碼並不擁有這個參照。如果該物件被銷毀，它會成為一個迷途指標 (dangling pointer)。例如，一次垃圾回收 (garbage collection) 可以移除對物件的最後一個 *strong reference*（強參照），而將該物件銷毀。

對 *borrowed reference* 呼叫 `Py_INCREF()` 以將它原地 (in-place) 轉換為 *strong reference* 是被建議的做法，除非該物件不能在最後一次使用借用參照之前被銷毀。`Py_NewRef()` 函式可用於建立一個新的 *strong reference*。

bytes-like object（類位元組串物件）

一個支援 bufferobjects 且能夠匯出 C-*contiguous* 緩衝區的物件。這包括所有的 `bytes`、`bytearray` 和 `array.array` 物件，以及許多常見的 `memoryview` 物件。類位元組串物件可用於處理二進位資料的各種運算；這些運算包括壓縮、儲存至二進位檔案和透過 socket（插座）發送。

有些運算需要二進位資料是可變的。說明文件通常會將這些物件稱為「可讀寫的類位元組串物件」。可變緩衝區的物件包括 `bytearray`，以及 `bytearray` 的 `memoryview`。其他的運算需要讓二進位資料被儲存在不可變物件（「唯讀的類位元組串物件」）中；這些物件包括 `bytes`，以及 `bytes` 物件的 `memoryview`。

bytecode（位元組碼）

Python 的原始碼會被編譯成位元組碼，它是 Python 程式在 CPython 直譯器中的內部表示法。該位元組碼也會被暫存在 `.pyc` 檔案中，以便第二次執行同一個檔案時能夠更快速（可以不用從原始碼重新編譯出位元組碼）。這種「中間語言 (intermediate language)」據說是運行在一個 *virtual machine*（虛擬機器）上，該虛擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python 虛擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在 `dis` 模組的說明文件中找到。

callable（可呼叫物件）

一個 callable 是可以被呼叫的物件，呼叫時可能以下列形式帶有一組引數（請見 *argument*）：

```
callable(argument1, argument2, argumentN)
```

一個 *function* 與其延伸的 *method* 都是 callable。一個有實作 *__call__()* 方法的 class 之實例也是個 callable。

callback（回呼）

作為引數被傳遞的一個副程式 (subroutine) 函式，會在未來的某個時間點被執行。

class（類別）

一個用於建立使用者定義物件的模板。Class 的定義通常會包含 method 的定義，這些 method 可以在 class 的實例上進行操作。

class variable（類別變數）

一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

closure variable（閉包變數）

從外部作用域中定義且從巢狀作用域參照的自由變數，不是於 runtime 從全域或內建命名空間解析。可以使用 *nonlocal* 關鍵字明確定義以允許寫入存取，或者如果僅需讀取變數則隱式定義即可。

例如在下面程式碼中的 inner 函式中，x 和 print 都是自由變數，但只有 x 是閉包變數：

```python
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

由於 *codeobject.co_freevars* 屬性（儘管名稱如此，但它僅包含閉包變數的名稱，而不是列出所有參照的自由變數），當預期含義是特指閉包變數時，有時候甚至也會使用更通用的自由變數一詞。

complex number（複數）

一個我們熟悉的實數系統的擴充，在此所有數字都會被表示為一個實部和一個虛部之和。複數就是虛數單位（-1 的平方根）的實數倍，此單位通常在數學中被寫為 i，在工程學中被寫為 j。Python 內建了對複數的支援，它是用後者的記法來表示複數；虛部會帶著一個後綴的 j 被編寫，例如 3+1j。若要將 math 模組裡的工具等效地用於複數，請使用 cmath 模組。複數的使用是一個相當進階的數學功能。如果你沒有察覺到對它們的需求，那你幾乎能確定你可以安全地忽略它們。

context（情境）

This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a *with* statement.

- The collection of keyvalue bindings associated with a particular contextvars.Context object and accessed via ContextVar objects. Also see *context variable*.

- 一個 contextvars.Context 物件。另請參閱 *current context*。

context management protocol（情境管理協定）

由 *with* 陳述式所呼叫的 *__enter__()* 和 *__exit__()* 方法。另請參閱 **PEP 343**。

context manager（情境管理器）

An object which implements the *context management protocol* and controls the environment seen in a *with* statement. See **PEP 343**.

context variable（情境變數）

A variable whose value depends on which context is the *current context*. Values are accessed via contextvars.ContextVar objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous（連續的）

如果一個緩衝區是 *C-contiguous* 或是 *Fortran contiguous*，則它會確切地被視為是連續的。零維 (zerodimensional) 的緩衝區都是 C 及 Fortran contiguous。在一維 (one-dimensional) 陣列中，各項目必須在記憶體中彼此相鄰地排列，而其索引順序是從零開始遞增。在多維的 (multidimensional) C-contiguous 陣列中，按記憶體位址的順序訪問各個項目時，最後一個索引的變化最快。然而，在 Fortran contiguous 陣列中，第一個索引的變化最快。

coroutine（協程）

協程是副程式 (subroutine) 的一種更為廣義的形式。副程式是在某個時間點被進入並在另一個時間點被退出。協程可以在許多不同的時間點被進入、退出和回復。它們能夠以 *async def* 陳述式被實作。另請參閱 **PEP 492**。

coroutine function（協程函式）

一個回傳 *coroutine*（協程）物件的函式。一個協程函式能以 *async def* 陳述式被定義，並可能會包含 *await*、*async for* 和 *async with* 關鍵字。這些關鍵字由 **PEP 492** 引入。

CPython

Python 程式語言的標準實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 IronPython。

current context

The *context* (contextvars.Context object) that is currently used by ContextVar objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see asyncio) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

decorator（裝飾器）

一個函式，它會回傳另一個函式，通常它會使用 @wrapper 語法，被應用為一種函式的變換 (function transformation)。裝飾器的常見範例是 classmethod() 和 staticmethod()。

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

```
def f(arg):
    ...
f = staticmethod(f)
```

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

Class 也存在相同的概念，但在那裡比較不常用。關於裝飾器的更多內容，請參閱*函式定義*和*class
定義*的說明文件。

descriptor（描述器）

任何定義了`__get__()`、`__set__()`或`__delete__()`method 的物件。當一個 class 屬性是一個描述
器時，它的特殊連結行為會在屬性查找時被觸發。通常，使用 *a.b* 來取得、設定或刪除某個屬性時，
會在 *a* 的 class 字典中查找名稱為 *b* 的物件，但如果 *b* 是一個描述器，則相對應的描述器 method 會
被呼叫。對描述器的理解是深入理解 Python 的關鍵，因為它們是許多功能的基礎，這些功能包括
函式、method、屬性 (property)、class method、靜態 method，以及對 super class（父類別）的參照。

關於描述器 method 的更多資訊，請參閱*實作描述器*或描述器使用指南。

dictionary（字典）

一個關聯陣列 (associative array)，其中任意的鍵會被對映到值。鍵可以是任何帶有`__hash__()`
和`__eq__()` method 的物件。在 Perl 中被稱為雜湊 (hash)。

dictionary comprehension（字典綜合運算）

一種緊密的方法，用來處理一個可迭代物件中的全部或部分元素，並將處理結果以一個字典回傳。
`results = {n: n ** 2 for n in range(10)}`會產生一個字典，它包含了鍵 n 對映到值 n **
2。請參閱*Displays for lists, sets and dictionaries*。

dictionary view（字典檢視）

從`dict.keys()`、`dict.values()`及`dict.items()`回傳的物件被稱為字典檢視。它們提供了字
典中項目的動態檢視，這表示當字典有變動時，該檢視會反映這些變動。若要強制將字典檢視轉為
完整的 list（串列），須使用`list(dictview)`。請參閱 dict-views。

docstring（說明字串）

一個在 class、函式或模組中，作為第一個運算式出現的字串文本。雖然它在套件執行時會被忽略，
但它會被編譯器辨識，並被放入所屬 class、函式或模組的`__doc__`屬性中。由於說明字串可以透
過內省 (introspection) 來瀏覽，因此它是物件的說明文件存放的標準位置。

duck-typing（鴨子型別）

一種程式設計風格，它不是藉由檢查一個物件的型別來確定它是否具有正確的介面；取而代之的是，
method 或屬性會單純地被呼叫或使用。（「如果它看起來像一隻鴨子而且叫起來像一隻鴨子，那麼
它一定是一隻鴨子。」）因強調介面而非特定型別，精心設計的程式碼能讓多形替代 (polymorphic
substitution) 來增進它的靈活性。鴨子型別要避免使用`type()`或`isinstance()`進行測試。（但是請
注意，鴨子型別可以用*抽象基底類別 (abstract base class)* 來補充。）然而，它通常會使用`hasattr()`
測試，或是*EAFP* 程式設計風格。

EAFP

Easier to ask for forgiveness than permission.（請求寬恕比請求許可更容易。）這種常見的 Python 編碼
風格會先假設有效的鍵或屬性的存在，並在該假設被推翻時再捕獲例外。這種乾淨且快速的風格，
其特色是存在許多的`try`和`except`陳述式。該技術與許多其他語言（例如 C）常見的*LBYL* 風格形
成了對比。

expression（運算式）

一段可以被評估並求值的語法。換句話說，一個運算式就是文字、名稱、屬性存取、運算子或函式
呼叫等運算式元件的累積，而這些元件都能回傳一個值。與許多其他語言不同的是，並非所有的
Python 語言構造都是運算式。另外有一些*statement*（陳述式）不能被用作運算式，例如`while`。賦
值 (assignment) 也是陳述式，而不是運算式。

extension module（擴充模組）

一個以 C 或 C++ 編寫的模組，它使用 Python 的 C API 來與核心及使用者程式碼進行互動。

f-string（f 字串）

以`'f'`或`'F'`為前綴的字串文本通常被稱為「f 字串」，它是*格式化的字串文本*的縮寫。另請參閱
PEP 498。

file object（檔案物件）

一個讓使用者透過檔案導向 (file-oriented) API（如 read() 或 write() 等 method）來操作底層資源的物件。根據檔案物件被建立的方式，它能夠協調對真實磁碟檔案或是其他類型的儲存器或通訊裝置（例如標准輸入／輸出、記憶體中緩衝區、socket（插座）、管線 (pipe) 等）的存取。檔案物件也被稱為類檔案物件 *(file-like object)* 或串流 *(stream)*。

實際上，有三種檔案物件：原始的二進位檔案、緩衝的二進位檔案和文字檔案。它們的介面在 io 模組中被定義。建立檔案物件的標准方法是使用 open() 函式。

file-like object（類檔案物件）

file object（檔案物件）的同義字。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）

Python 所使用的一種編碼和錯誤處理函式，用來解碼來自作業系統的位元組，以及將 Unicode 編碼到作業系統。

檔案系統編碼必須保證能成功解碼所有小於 128 的位元組。如果檔案系統編碼無法提供此保證，則 API 函式會引發 UnicodeError。

sys.getfilesystemencoding() 和 sys.getfilesystemencodeerrors() 函式可用於取得檔案系統編碼和錯誤處理函式。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）會在 Python 啟動時由 PyConfig_Read() 函式來配置：請參閱 filesystem_encoding，以及 PyConfig 的成員 filesystem_errors。

另請參閱*locale encoding*（區域編碼）。

finder（尋檢器）

一個物件，它會嘗試為正在被 import 的模組尋找*loader*（載入器）。

有兩種類型的尋檢器：*元路徑尋檢器 (meta path finder)* 會使用 sys.meta_path，而*路徑項目尋檢器 (path entry finder)* 會使用 sys.path_hooks。

請參閱*尋檢器 (Finder) 與載入器 (Loader)* 和 importlib 以了解更多細節。

floor division（向下取整除法）

向下無條件舍去到最接近整數的數學除法。向下取整除法的運算子是 //。例如，運算式 11 // 4 的計算結果為 2，與 float（浮點數）真除法所回傳的 2.75 不同。請注意，(-11) // 4 的結果是 -3，因為是 -2.75 被向下無條件舍去。請參閱 **PEP 238**。

free threading（自由執行緒）

為一種執行緒模型，多個執行緒可以在同一直譯器中同時運行 Python 位元組碼。這與*全域直譯器鎖*形成對比，後者一次只允許一個執行緒執行 Python 位元組碼。請參閱 **PEP 703**。

free variable（自由變數）

Formally, as defined in the *language execution model*, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the *codeobject.co_freevars* attribute, the term is also sometimes used as a synonym for *closure variable*.

function（函式）

一連串的陳述式，它能夠向呼叫者回傳一些值。它也可以被傳遞零個或多個*引數*，這些引數可被使用於函式本體的執行。另請參閱*parameter*（參數）、*method*（方法），以及*函式定義*章節。

function annotation（函式註釋）

函式參數或回傳值的一個*annotation*（註釋）。

函式註釋通常被使用於*型別提示*：例如，這個函式預期會得到兩個 int 引數，並會有一個 int 回傳值：

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

函式註釋的語法在*函式定義*章節有詳細解釋。

請參閱*variable annotation* 和 **PEP 484**，皆有此功能的描述。關於註釋的最佳實踐方法，另請參閱 annotations-howto。

__future__

> *future* 陳述式: from __future__ import <feature>，會指示編譯器使用那些在 Python 未來的發布版本中將成為標準的語法或語義，來編譯當前的模組。而 __future__ 模組則記錄了 *feature*（功能）可能的值。透過 import 此模組並對其變數求值，你可以看見一個新的功能是何時首次被新增到此語言中，以及它何時將會（或已經）成為預設的功能:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection（垃圾回收）

> 當記憶體不再被使用時，將其釋放的過程。Python 執行垃圾回收，是透過參照計數 (reference counting)，以及一個能夠檢測和中斷參照循環 (reference cycle) 的循環垃圾回收器 (cyclic garbage collector) 來完成。垃圾回收器可以使用 gc 模組對其進行控制。

generator（產生器）

> 一個會回傳 *generator iterator*（產生器迭代器）的函式。它看起來像一個正常的函式，但不同的是它包含了 yield 運算式，能產生一系列的值，這些值可用於 for 迴圈，或是以 next() 函式，每次檢索其中的一個值。

> 這個術語通常用來表示一個產生器函式，但在某些情境中，也可能是表示產生器迭代器。萬一想表達的意思不夠清楚，那就使用完整的術語，以避免歧義。

generator iterator（產生器迭代器）

> 一個由 *generator*（產生器）函式所建立的物件。

> 每個 yield 會暫停處理程序，並記住執行狀態（包括區域變數及擱置中的 try 陳述式）。當產生器迭代器回復時，它會從停止的地方繼續執行（與那些每次調用時都要重新開始的函式有所不同）。

generator expression（產生器運算式）

> 一個會回傳迭代器的運算式。它看起來像一個正常的運算式，後面接著一個 for 子句，該子句定義了迴圈變數、範圍以及一個選擇性的 if 子句。該組合運算式會為外層函式產生多個值:

```
>>> sum(i*i for i in range(10))         # 平方之和 0, 1, 4, ... 81
285
```

generic function（泛型函式）

> 一個由多個函式組成的函式，該函式會對不同的型別實作相同的運算。呼叫期間應該使用哪種實作，是由調度演算法 (dispatch algorithm) 來決定。

> 另請參考 *single dispatch*（單一調度）術語表條目、functools.singledispatch() 裝飾器和 **PEP 443**。

generic type（泛型型別）

> 一個能夠被參數化 (parameterized) 的 *type*（型別）; 通常是一個容器型別，像是 list 和 dict。它被用於型別提示和註釋。

> 詳情請參考泛型別名型別、**PEP 483**、**PEP 484**、**PEP 585** 和 typing 模組。

GIL

> 請參考 *global interpreter lock*（全域直譯器鎖）。

global interpreter lock（全域直譯器鎖）

> *CPython* 直譯器所使用的機制，用以確保每次都只有一個執行緒能執行 Python 的 *bytecode*（位元組碼）。透過使物件模型（包括關鍵的內建型別，如 dict）自動地避免並行存取 (concurrent access) 的危險，此機制可以簡化 CPython 的實作。鎖定整個直譯器，會使直譯器更容易成為多執行緒 (multi-threaded)，但代價是會犧牲掉多處理器的機器能夠提供的一大部分平行性 (parallelism)。

> 然而，有些擴充模組，無論是標準的或是第三方的，它們被設計成在執行壓縮或雜湊等計算密集 (computationally intensive) 的任務時，可以解除 GIL。另外，在執行 I/O 時，GIL 總是會被解除。

> 從 Python 3.13 開始可以使用 --disable-gil 建置設定來停用 GIL。使用此選項建立 Python 後，必須使用 -X gil=0 來執行程式碼，或者設定 PYTHON_GIL=0 環境變數後再執行程式碼。此功能可以提高多執行緒應用程式的效能，並使多核心 CPU 的高效使用變得更加容易。有關更多詳細資訊，請參考 **PEP 703**。

hash-based pyc（雜⬚架構的 pyc）

一個位元組碼 (bytecode) 暫存檔，它使用雜⬚值而不是對應原始檔案的最後修改時間，來確定其有效性。請參⬚*Cached bytecode invalidation*。

hashable（可雜⬚的）

如果一個物件有一個雜⬚值，該值在其生命⬚期中永不改變（它需要一個`__hash__()` method），且可與其他物件互相比較（它需要一個`__eq__()` method），那⬚它就是一個可雜⬚物件。比較結果⬚相等的多個可雜⬚物件，它們必須擁有相同的雜⬚值。

可雜⬚性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因⬚這些資料結構都在其⬚部使用了雜⬚值。

大多數的 Python 不可變⬚建物件都是可雜⬚的；可變的容器（例如 list 或 dictionary）⬚不是；而不可變的容器（例如 tuple（元組）和 frozenset），只有當它們的元素是可雜⬚的，它們本身才是可雜⬚的。若物件是使用者自定 class 的實例，則這些物件會被預設⬚可雜⬚的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜⬚值則是衍生自它們的 id()。

IDLE

Python 的 Integrated Development and Learning Environment（整合開發與學習環境）。idle 是一個基本的編輯器和直譯器環境，它和 Python 的標准發行版本一起被提供。

immortal（不滅）

不滅物件 *(Immortal objects)* 是 **PEP 683** 引入的 CPython 實作細節。

如果一個物件是不滅的，它的參照計數永遠不會被修改，因此在直譯器運行時它永遠不會被釋放。例如，True 和 None 在 CPython 中是不滅的。

immutable（不可變物件）

一個具有固定值的物件。不可變物件包括數字、字串和 tuple（元組）。這類物件是不能被改變的。如果一個不同的值必須被儲存，則必須建立一個新的物件。它們在需要⬚定雜⬚值的地方，扮演重要的角色，例如 dictionary（字典）中的一個鍵。

import path（引入路徑）

一個位置（或路徑項目）的列表，而那些位置就是在 import 模組時，會被*path based finder*（基於路徑的尋檢器）搜尋模組的位置。在 import 期間，此位置列表通常是來自 sys.path，但對於子套件 (subpackage) 而言，它也可能是來自父套件的 __path__ 屬性。

importing（引入）

一個過程。一個模組中的 Python 程式碼可以透過此過程，被另一個模組中的 Python 程式碼使用。

importer（引入器）

一個能⬚尋找及載入模組的物件；它既是*finder*（尋檢器）也是*loader*（載入器）物件。

interactive（互動的）

Python 有一個互動式直譯器，這表示你可以在直譯器的提示字元輸入陳述式和運算式，立即執行它們⬚且看到它們的結果。只要⬚動 python，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模塊和包的非常⬚大的方法（請記住 help(x)）。更多互動式模式相關資訊請見 tut-interac。

interpreted（直譯的）

Python 是一種直譯語言，而不是編譯語言，不過這個區分可能有些模糊，因⬚有位元組碼 (bytecode) 編譯器的存在。這表示原始檔案可以直接被運行，而不需明確地建立另一個執行檔，然後再執行它。直譯語言通常比編譯語言有更短的開發／除錯⬚期，不過它們的程式通常也運行得較慢。另請參⬚*interactive*（互動的）。

interpreter shutdown（直譯器關閉）

當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵⬚部結構。它也會多次呼叫垃圾回收器 *(garbage collector)*。這能⬚觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，⬚執行其中的程式碼。在關閉階段被執行的程式碼會遇到各種例外，因⬚它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

直譯器關閉的主要原因，是 __main__ 模組或正被運行的⬚本已經執行完成。

iterable（可⬚代物件）

An object capable of returning its members one at a time. Examples of iterables include all sequence types

(such as list, str, and tuple) and some non-sequence types like dict, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

可迭代物件可用於 *for* 迴圈和許多其他需要一個序列的地方 (zip()、map()...)。當一個可迭代物件作為引數被傳遞給內建函式 iter() 時，它會將該物件回傳一個迭代器。此迭代器適用於針對一組值進行一遍 (one pass) 運算。使用迭代器時，通常不一定要呼叫 iter() 或自行處理迭代器物件。*for* 陳述式會自動地為你處理這些事，它會建立一個暫時性的未命名變數，用於在迴圈期間保有該迭代器。另請參閱 *iterator*（迭代器）、*sequence*（序列）和 *generator*（產生器）。

iterator（迭代器）

一個表示資料流的物件。重複地呼叫迭代器的 `__next__()` method（或是將它傳遞給內建函式 next()）會依序回傳資料流中的各項目。當不再有資料時，則會引發 StopIteration 例外。此時，該迭代器物件已被用盡，而任何對其 `__next__()` method 的進一步呼叫，都只會再次引發 StopIteration。迭代器必須有一個 `__iter__()` method，它會回傳迭代器物件本身，所以每個迭代器也都是可迭代物件，且可以用於大多數適用其他可迭代物件的場合。一個明顯的例外，是嘗試多遍迭代 (multiple iteration passes) 的程式碼。一個容器物件（像是 list）在每次你將它傳遞給 iter() 函式或在 *for* 迴圈中使用它時，都會產生一個全新的迭代器。使用迭代器嘗試此事（多遍迭代）時，只會回傳在前一遍迭代中被用過的、同一個已被用盡的迭代器物件，使其看起來就像一個空的容器。

在 typeiter 文中可以找到更多資訊。

CPython 並不是始終如一地都會檢查「迭代器有定義 `__iter__()`」這個規定。另請注意，free-threading（自由執行緒）CPython 不保證迭代器操作的執行緒安全。

key function（鍵函式）

鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，locale.strxfrm() 被用來產生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括 min()、max()、sorted()、list.sort()、heapq.merge()、heapq.nsmallest()、heapq.nlargest() 和 itertools.groupby()。

有幾種方法可以建立一個鍵函式。例如，str.lower() method 可以作為不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 *lambda* 運算式被建造，例如 lambda r: (r[0], r[2])。另外，operator.attrgetter()、operator.itemgetter() 和 operator.methodcaller() 是三個鍵函式的建構函式 (constructor)。關於如何建立和使用鍵函式的範例，請參閱如何排序。

keyword argument（關鍵字引數）

請參閱 *argument*（引數）。

lambda

由單一 *expression*（運算式）所組成的一個匿名行內函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 lambda [parameters]: expression

LBYL

Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先決條件。這種風格與 *EAFP* 方式形成對比，且它的特色是會有許多 *if* 陳述式的存在。

在一個多執行緒環境中，LBYL 方式有在「三思」和「後行」之間引入了競爭條件 (race condition) 的風險。例如以下程式碼 if key in mapping: return mapping[key]，如果另一個執行緒在測試之後但在查找之前，從 *mapping* 中移除了 *key*，則該程式碼就會失效。這個問題可以用鎖 (lock) 或使用 EAFP 編碼方式來解決。

list（串列）

一個 Python 內建的 *sequence*（序列）。儘管它的名字是 list，它其實更類似其他語言中的一個陣列 (array) 而較不像一個鏈結串列 (linked list)，因為存取元素的時間複雜度是 $O(1)$。

list comprehension（串列綜合運算）

一種用來處理一個序列中的全部或部分元素，並將處理結果以一個 list 回傳的簡要方法。result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0] 會產生一個字串 list，其中包含 0 到 255 範圍內，所有偶數的十六進位數 (0x..)。*if* 子句是選擇性的。如果省略它，則 range(256) 中的所有元素都會被處理。

loader（載入器）

一個能夠載入模組的物件。它必須定義 exec_module() 和 create_module() 方法以實作 Loader 介面。載入器通常是被*finder*（尋檢器）回傳。更多細節請參閱：

- *尋檢器 (Finder) 與載入器 (Loader)*
- importlib.abc.Loader
- **PEP 302**

locale encoding（區域編碼）

在 Unix 上，它是 LC_CTYPE 區域設定的編碼。它可以用 locale.setlocale(locale.LC_CTYPE, new_locale) 來設定。

在 Windows 上，它是 ANSI 代碼頁（code page，例如 "cp1252"）。

在 Android 和 VxWorks 上，Python 使用 "utf-8" 作為區域編碼。

locale.getencoding() 可以用來取得區域編碼。

也請參考*filesystem encoding and error handler*。

magic method（魔術方法）

special method（特殊方法）的一個非正式同義詞。

mapping（對映）

一個容器物件，它支援任意鍵的查找，且能實作 abstract base classes（抽象基底類別）中，collections.abc.Mapping 或 collections.abc.MutableMapping 所指定的 method。範例包括 dict、collections.defaultdict、collections.OrderedDict 和 collections.Counter。

meta path finder（元路徑尋檢器）

一種經由搜尋 sys.meta_path 而回傳的*finder*（尋檢器）。元路徑尋檢器與*路徑項目尋檢器 (path entry finder)* 相關但是不同。

關於元路徑尋檢器實作的 method，請參閱 importlib.abc.MetaPathFinder。

metaclass（元類別）

一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個 base class（基底類別）的列表。Metaclass 負責接受這三個引數，並建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特別之處在於它能夠建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供強大且優雅的解決方案。它們已被用於記錄屬性存取、增加執行緒安全性、追蹤物件建立、實作單例模式 (singleton)，以及許多其他的任務。

更多資訊可以在*Metaclasses*章節中找到。

method（方法）

一個在 class 本體內被定義的函式。如果 method 作為其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成為它的第一個*argument*（引數）（此引數通常被稱為 self）。請參閱*function*（函式）和*nested scope*（巢狀作用域）。

method resolution order（方法解析順序）

方法解析順序是在查找某個成員的過程中，base class（基底類別）被搜尋的順序。關於 Python 自 2.3 版直譯器所使用的演算法細節，請參閱 python_2.3_mro。

module（模組）

一個擔任 Python 程式碼的組織單位 (organizational unit) 的物件。模組有一個命名空間，它包含任意的 Python 物件。模組是藉由*importing* 的過程，被載入至 Python。

另請參閱*package*（套件）。

module spec（模組規格）

一個命名空間，它包含用於載入模組的 import 相關資訊。它是 importlib.machinery.ModuleSpec 的一個實例。

另請參閱模組規格。

MRO

請參閱*method resolution order*（方法解析順序）。

mutable（可變物件）
可變物件可以改變它們的值，但維持它們的 `id()`。另請參⯑*immutable*（不可變物件）。

named tuple（附名元組）
術語「named tuple（附名元組）」是指從 tuple 繼承的任何型⯑或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型⯑或 class 也可以具有其他的特性。

有些⯑建型⯑是 named tuple，包括由 `time.localtime()` 和 `os.stat()` 回傳的值。另一個例子是 `sys.float_info`:

```
>>> sys.float_info[1]                 # indexed access
1024
>>> sys.float_info.max_exp            # named field access
1024
>>> isinstance(sys.float_info, tuple) # kind of tuple
True
```

有些 named tuple 是⯑建型⯑（如上例）。或者，一個 named tuple 也可以從一個正規的 class 定義來建立，只要該 class 是繼承自 `tuple`，且定義了附名欄位 (named field) 即可。這類的 class 可以手工編寫、可以繼承自 `typing.NamedTuple` 來建立，也可以使用工廠函式 (factory function) `collections.namedtuple()` 來建立。後者技術也增加了一些額外的 method，這些 method 可能是在手寫或⯑建的 named tuple 中，無法找到的。

namespace（命名空間）
變數被儲存的地方。命名空間是以 dictionary（字典）被實作。有區域的、全域的及⯑建的命名空間，而在物件中（在 method 中）也有巢狀的命名空間。命名空間藉由防止命名衝突，來支援模組化。例如，函式 `builtins.open` 和 `os.open()` 是透過它們的命名空間來區分彼此。命名空間也藉由明確地區分是哪個模組在實作一個函式，來增進可讀性及可維護性。例如，寫出 `random.seed()` 或 `itertools.islice()` 明確地表示，這些函式分⯑是由 `random` 和 `itertools` 模組在實作。

namespace package（命名空間套件）
A *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and 命名空間套件.

另請參⯑*module*（模組）。

nested scope（巢狀作用域）
能⯑參照外層定義 (enclosing definition) 中的變數的能力。舉例來⯑，一個函式如果是在另一個函式中被定義，則它便能⯑參照外層函式中的變數。請注意，在預設情⯑下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最⯑層作用域中讀取及寫入。同樣地，全域變數是在全域命名空間中讀取及寫入。`nonlocal` 容許對外層作用域進行寫入。

new-style class（新式類⯑）
一個舊名，它是指現在所有的 class 物件所使用的 class 風格。在早期的 Python 版本中，只有新式 class 才能使用 Python 較新的、多樣的功能，像是`__slots__`、描述器 (descriptor)、屬性 (property)、`__getattribute__()`、class method（類⯑方法）和 static method（⯑態方法）。

object（物件）
具有狀態（屬性或值）及被定義的行⯑（method）的任何資料。它也是任何*new-style class*（新式類⯑）的最終 base class（基底類⯑）。

optimized scope（最佳化作用域）
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package（套件）
一個 Python 的*module*（模組），它可以包含子模組 (submodule) 或是遞⯑的子套件 (subpackage)。技

術上而言，套件就是具有 `__path__` 屬性的一個 Python 模組。

另請參⿰*regular package*（正規套件）和*namespace package*（命名空間套件）。

parameter（參數）

在*function*（函式）或 method 定義中的一個命名實體 (named entity)，它指明該函式能⿰接受的一個*argument*（引數），或在某些情⿰下指示多個引數。共有有五種不同的參數類型：

- *positional-or-keyword*（位置或關鍵字）：指明一個可以[按照位置](#)或是作⿰[關鍵字引數](#)被傳遞的引數。這是參數的預設類型，例如以下的 *foo* 和 *bar*：

```
def func(foo, bar=None): ...
```

- *positional-only*（僅限位置）：指明一個只能按照位置被提供的引數。在函式定義的參數列表中包含一個 / 字元，就可以在該字元前面定義僅限位置參數，例如以下的 *posonly1* 和 *posonly2*：

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*（僅限關鍵字）：指明一個只能以關鍵字被提供的引數。在函式定義的參數列表中，包含一個任意數量位置參數 (var-positional parameter) 或是單純的 * 字元，就可以在其後方定義僅限關鍵字參數，例如以下的 *kw_only1* 和 *kw_only2*：

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*（任意數量位置）：指明一串能以任意序列被提供的位置引數（在已被其他參數接受的任何位置引數之外）。這類參數是透過在其參數名稱字首加上 * 來定義的，例如以下的 *args*：

```
def func(*args, **kwargs): ...
```

- *var-keyword*（任意數量關鍵字）：指明可被提供的任意數量關鍵字引數（在已被其他參數接受的任何關鍵字引數之外）。這類參數是透過在其參數名稱字首加上 ** 來定義的，例如上面範例中的 *kwargs*。

參數可以指明引數是選擇性的或必需的，也可以⿰一些選擇性的引數指定預設值。

另請參⿰術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差⿰、`inspect.Parameter` class、[函式定義](#)章節，以及 **PEP 362**。

path entry（路徑項目）

在*import path*（引入路徑）中的一個位置，而*path based finder*（基於路徑的尋檢器）會參考該位置來尋找要 import 的模組。

path entry finder（路徑項目尋檢器）

被 `sys.path_hooks` 中的一個可呼叫物件 (callable)（意即一個*path entry hook*）所回傳的一種*finder*，它知道如何以一個*path entry*定位模組。

關於路徑項目尋檢器實作的 method，請參⿰ `importlib.abc.PathEntryFinder`。

path entry hook（路徑項目⿰）

在 `sys.path_hooks` 列表中的一個可呼叫物件 (callable)，若它知道如何在一個特定的*path entry* 中尋找模組，則會回傳一個*path entry finder*（路徑項目尋檢器）。

path based finder（基於路徑的尋檢器）

預設的[元路徑尋檢器 *(meta path finder)*](#) 之一，它會在一個*import path* 中搜尋模組。

path-like object（類路徑物件）

一個表示檔案系統路徑的物件。類路徑物件可以是一個表示路徑的 `str` 或 `bytes` 物件，或是一個實作 `os.PathLike` 協定的物件。透過呼叫 `os.fspath()` 函式，一個支援 `os.PathLike` 協定的物件可以被轉⿰⿰ `str` 或 `bytes` 檔案系統路徑；而 `os.fsdecode()` 及 `os.fsencode()` 則分⿰可用於確保 `str` 及 `bytes` 的結果。由 **PEP 519** 引入。

PEP

Python Enhancement Proposal（Python 增⿰提案）。PEP 是一份設計⿰明文件，它能⿰ Python 社群提

供資訊，或是描述 Python 的一個新功能或該功能的程序和環境。PEP 應該要提供簡潔的技術規範以及被提案功能的運作原理。

PEP 的存在目的，是要成⊟重大新功能的提案、社群中關於某個問題的意見收集，以及已納入 Python 的設計⊟策的記⊟，這些過程的主要機制。PEP 的作者要負責在社群⊟建立共識⊟記⊟反對意見。

請參⊟ **PEP 1**。

portion（部分）

在單一目⊟中的一組檔案（也可能儲存在一個 zip 檔中），這些檔案能對一個命名空間套件 (namespace package) 有所貢獻，如同 **PEP 420** 中的定義。

positional argument（位置引數）

請參⊟*argument*（引數）。

provisional API（暫行 API）

暫行 API 是指，從標準函式庫的向後相容性 (backwards compatibility) 保證中，故意被排除的 API。雖然此類介面，只要它們被標示⊟暫行的，理論上⊟不會有重大的變更，但如果核心開發人員認⊟有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更⊟不會無端地⊟生——只有 API 被納入之前未察覺的嚴重基本缺陷被揭露時，它們才會發生。

即使對於暫行 API，向後不相容的變更也會被視⊟「最後的解⊟方案」——對於任何被發現的問題，仍然會盡可能找出一個向後相容的解⊟方案。

這個過程使得標準函式庫能隨著時間不斷進化，而避免耗費過長的時間去鎖定有問題的設計錯誤。請參⊟ **PEP 411** 了解更多細節。

provisional package（暫行套件）

請參⊟*provisional API*（暫行 API）。

Python 3000

Python 3.x 系列版本的⊟稱（很久以前創造的，當時第 3 版的發布是在⊟遠的未來。）也可以縮寫⊟「Py3k」。

Pythonic（Python 風格的）

一個想法或一段程式碼，它應用了 Python 語言最常見的慣用語，而不是使用其他語言常見的概念來實作程式碼。例如，Python 中常見的一種習慣用法，是使用一個 *for* 陳述式，對一個可⊟代物件的所有元素進行⊟圈。許多其他語言⊟⊟有這種類型的架構，所以不熟悉 Python 的人有時會使用一個數值計數器來代替：

```
for i in range(len(food)):
    print(food[i])
```

相較之下，以下方法更簡潔、更具有 Python 風格：

```
for piece in food:
    print(piece)
```

qualified name（限定名稱）

一個「點分隔名稱」，它顯示從一個模組的全域作用域到該模組中定義的 class、函式或 method 的「路徑」，如 **PEP 3155** 中的定義。對於頂層的函式和 class 而言，限定名稱與其物件名稱相同：

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

當用於引用模組時，完全限定名⿱ (fully qualified name) 是表示該模組的完整點分隔路徑，包括任何的父套件，例如 email.mime.text:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count（參照計數）

對於一個物件的參照次數。當一個物件的參照計數下降到零時，它會被解除配置 (deallocated)。有些物件是「不滅的 (immortal)」⿱擁有不會被改變的參照計數，也因此永遠不會被解除配置。參照計數通常在 Python 程式碼中看不到，但它⿱是*CPython* 實作的一個關鍵元素。程式設計師可以呼叫 getrefcount() 函式來回傳一個特定物件的參照計數。

regular package（正規套件）

一個傳統的*package*（套件），例如一個包含 __init__.py 檔案的目⿱。

另請參⿱*namespace package*（命名空間套件）。

REPL

「read-eval-print ⿱圈 (read–eval–print loop)」的縮寫，是互動式直譯器 shell 的另一個名稱。

__slots__

在 class ⿱部的一個宣告，它藉由預先宣告實例屬性的空間，以及消除實例 dictionary（字典），來節省記憶體。雖然該技術很普遍，但它有點難以正確地使用，最好保留給那種在一個記憶體關鍵 (memory-critical) 的應用程式中存在大量實例的罕見情⿱。

sequence（序列）

一個*iterable*（可⿱代物件），它透過*__getitem__()* special method（特殊方法），使用整數索引來支援高效率的元素存取，⿱定義了一個*__len__()* method 來回傳該序列的長度。一些⿱建序列型⿱包括 list、str、tuple 和 bytes。請注意，雖然 dict 也支援*__getitem__()* 和 __len__()，但它被視⿱對映 (mapping) 而不是序列，因⿱其查找方式是使用任意的*hashable* 鍵，而不是整數。

抽象基底類⿱ (abstract base class) collections.abc.Sequence 定義了一個更加豐富的介面，⿱不僅止於*__getitem__()* 和*__len__()*，還增加了 count()、index()、*__contains__()* 和*__reversed__()*。實作此擴充介面的型⿱，可以使用 register() 被明確地⿱⿱。更多關於序列方法的文件，請見常見序列操作。

set comprehension（集合綜合運算）

一種緊密的方法，用來處理一個可⿱代物件中的全部或部分元素，⿱將處理結果以一個 set 回傳。results = {c for c in 'abracadabra' if c not in 'abc'} 會⿱生一個字串 set: {'r', 'd'}。請參⿱*Displays for lists, sets and dictionaries*。

single dispatch（單一調度）

generic function（泛型函式）調度的一種形式，在此，實作的選擇是基於單一引數的型⿱。

slice（切片）

一個物件，它通常包含一段*sequence*（序列）的某一部分。建立一段切片的方法是使用下標符號 (subscript notation) []，若要給出多個數字，則在數字之間使用冒號，例如 variable_name[1:3:5]。在括號（下標）符號的⿱部，會使用 slice 物件。

soft deprecated（軟性⿱用）

被軟性⿱用的 API 代表不應再用於新程式碼中，但在現有程式碼中繼續使用它仍會是安全的。API 仍會以文件記⿱⿱會被測試，但不會被繼續改進。

與正常⿱用不同，軟性⿱用⿱有⿱除 API 的規劃，也不會發出警告。

請參⿱ PEP 387: 軟性⿱用。

special method（特殊方法）

一種會被 Python 自動呼叫的 method，用於對某種型⿱執行某種運算，例如加法。這種 method 的名稱會在開頭和結尾有兩個下底⿱。Special method 在*Special method names*中有詳細⿱明。

statement（陳述式）

陳述式是一個套組（suite，一個程式碼「區塊」）中的一部分。陳述式可以是一個*expression*（運算式），或是含有關鍵字（例如*if*、*while* 或*for*）的多種結構之一。

static type checker（囗態型囗檢查器）

會讀取 Python 程式碼囗分析的外部工具，能囗找出錯誤，像是使用了不正確的型囗。另請參囗*型囗提示 (type hints)* 以及 typing 模組。

strong reference（囗參照）

在 Python 的 C API 中，囗參照是對物件的參照，該物件囗持有該參照的程式碼所擁有。建立參照時透過呼叫 Py_INCREF() 來獲得囗參照、囗除參照時透過 Py_DECREF() 釋放囗參照。

Py_NewRef() 函式可用於建立一個對物件的囗參照。通常，在退出囗參照的作用域之前，必須在該囗參照上呼叫 Py_DECREF() 函式，以避免囗漏一個參照。

另請參囗*borrowed reference*（借用參照）。

text encoding（文字編碼）

Python 中的字串是一個 Unicode 碼點 (code point) 的序列（範圍在 U+0000 -- U+10FFFF 之間）。若要儲存或傳送一個字串，它必須被序列化囗一個位元組序列。

將一個字串序列化囗位元組序列，稱囗「編碼」，而從位元組序列重新建立該字串則稱囗「解碼 (decoding)」。

有多種不同的文字序列化編解碼器 (codecs)，它們被統稱囗「文字編碼」。

text file（文字檔案）

一個能囗讀取和寫入 str 物件的一個*file object*（檔案物件）。通常，文字檔案實際上是存取位元組導向的資料流 (byte-oriented datastream) 囗會自動處理*text encoding*（文字編碼）。文字檔案的例子有：以文字模式（'r' 或 'w'）開囗的檔案、sys.stdin、sys.stdout 以及 io.StringIO 的實例。

另請參囗*binary file*（二進位檔案），它是一個能囗讀取和寫入*類位元組串物件 (bytes-like object)* 的檔案物件。

triple-quoted string（三引號囗字串）

由三個雙引號 (") 或單引號 (') 的作囗邊界的一個字串。雖然它們囗囗有提供囗於單引號字串的任何額外功能，但基於許多原因，它們仍是很有用的。它們讓你可以在字串中包含未跳囗 (unescaped) 的單引號和雙引號，而且它們不需使用連續字元 (continuation character) 就可以跨越多行，這使得它們在編寫囗明字串時特囗有用。

type（型囗）

一個 Python 物件的型囗囗定了它是什囗類型的物件；每個物件都有一個型囗。一個物件的型囗可以用它的 __*class*__ 屬性來存取，或以 type(obj) 來檢索。

type alias（型囗囗名）

一個型囗的同義詞，透過將型囗指定給一個識囗符 (identifier) 來建立。

型囗囗名對於簡化*型囗提示 (type hint)* 很有用。例如：

```python
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以寫成這樣，更具有可讀性：

```python
Color = tuple[int, int, int]

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

請參囗 typing 和 **PEP 484**，有此功能的描述。

type hint（型囗提示）

一種*annotation*（囗釋），它指定一個變數、一個 class 屬性或一個函式的參數或回傳值的預期型囗。

型囗提示是選擇性的，而不是被 Python 囗制的，但它們對囗態型囗檢查器 *(static type checkers)*很有用，囗能協助 IDE 完成程式碼的補全 (completion) 和重構 (refactoring)。

全域變數、class 屬性和函式（不含區域變數）的型囗提示，都可以使用 typing.get_type_hints() 來存取。

請參閱 typing 和 **PEP 484**，有此功能的描述。

universal newlines（通用換行字元）

一種解譯文字流 (text stream) 的方式，會將以下所有的情況識別為一行的結束：Unix 行尾慣例 '\n'、Windows 慣例 '\r\n' 和舊的 Macintosh 慣例 '\r'。請參閱 **PEP 278** 和 **PEP 3116**，以及用於 bytes.splitlines() 的附加用途。

variable annotation（變數註釋）

一個變數或 class 屬性的*annotation*（註釋）。

註釋變數或 class 屬性時，賦值是選擇性的：

```
class C:
    field: 'annotation'
```

變數註釋通常用於*型別提示 (type hint)*：例如，這個變數預期會取得 int（整數）值：

```
count: int = 0
```

變數註釋的語法在*Annotated assignment statements*章節有詳細的解釋。

請參閱*function annotation*（函式註釋）、**PEP 484** 和 **PEP 526**，皆有此功能的描述。關於註釋的最佳實踐方法，另請參閱 annotations-howto。

virtual environment（虛擬環境）

一個協作隔離 (cooperatively isolated) 的執行環境，能讓 Python 的使用者和應用程式得以安裝和升級 Python 發行套件，而不會對同一個系統上運行的其他 Python 應用程式的行為產生干擾。

另請參閱 venv。

virtual machine（虛擬機器）

一部完全由軟體所定義的電腦 (computer)。Python 的虛擬機器會執行由*bytecode*（位元組碼）編譯器所發出的位元組碼。

Zen of Python（Python 之禪）

Python 設計原則與哲學的列表，其內容有助於理解和使用此語言。此列表可以透過在互動式提式字元後輸入「import this」來找到它。

關於這份Ｆ明文件

Python Ｆ明文件是透過使用 Sphinx（一個原Ｆ Python 而生的文件Ｆ生器、目前是以獨立專案形式來維護）將使用 reStructuredText 撰寫的原始檔轉Ｆ而成。

如同 Python 自身，透過自願者的努力下Ｆ出文件與封裝後自動化執行工具。若想要回報臭蟲，請見 reporting-bugs 頁面，Ｆ含相關資訊。我們永遠歡迎新的自願者加入!

致謝:

- Fred L. Drake, Jr.，原始 Python 文件工具集的創造者以及一大部份Ｆ容的作者;

- 創造 reStructuredText 和 Docutils 工具組的 Docutils 專案;

- Fredrik Lundh 先生，Sphinx 從他的 Alternative Python Reference 計劃中獲得許多的好主意。

B.1 Python 文件的貢獻者們

許多人都曾Ｆ Python 這門語言、Python 標准函式庫和 Python Ｆ明文件貢獻過。Python 所發Ｆ的原始碼中含有部份貢獻者的清單，請見 Misc/ACKS 。

正因Ｆ Python 社群的撰寫與貢獻才有這份這Ｆ棒的Ｆ明文件 -- 感謝所有貢獻過的人們!

沿革與授權

C.1 軟體沿革

Python 是由荷蘭數學和計算機科學研究學會（CWI，見 https://www.cwi.nl）的 Guido van Rossum 於 1990 年代早期所創造，目的是作Ｆ一種稱Ｆ ABC 語言的後繼者。Ｆ管 Python 包含了許多來自其他人的貢獻，Guido 仍是其主要作者。

1995 年，Guido 在維吉尼亞州雷斯頓的國家創新研究公司（CNRI，見 https://www.cnri.reston.va.us）繼續他在 Python 的工作，Ｆ在那Ｆ發Ｆ了該軟體的多個版本。

2000 年五月，Guido 和 Python 核心開發團隊轉移到 BeOpen.com Ｆ成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations，後來成Ｆ Zope Corporation。2001 年，Python 軟體基金會（PSF，見 https://www.python.org/psf/）成立，這是一個專Ｆ擁有 Python 相關的智慧財Ｆ權而創立的非營利組織。Zope Corporation 過去是 PSF 的一個贊助會員。

所有的 Python 版本都是開源的（有關開源的定義，參Ｆ https://opensource.org）。歷史上，大多數但非全部的 Python 版本，也是 GPL 相容的；以下表格總結各個版本的差Ｆ。

發Ｆ版本	源自	年份	擁有者	GPL 相容性？(1)
0.9.0 至 1.2	不適用	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	是 (2)
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 以上	2.1.1	2001 至今	PSF	是

> ℹ️ 備Ｆ
>
> (1) GPL 相容Ｆ不表示我們是在 GPL 下發Ｆ Python。不像 GPL，所有的 Python 授權都可以讓你發Ｆ修改後的版本，但不一定要使你的變更成Ｆ開源。GPL 相容的授權使得 Python 可以結合其他

在 GPL 下發⬚的軟體一起使用；但其它的授權則不行。

(2) 根據 Richard Stallman 的⬚法，1.6.1 不是 GPL 相容的，因⬚其授權有一個法律選擇條款。然而根據 CNRI 的⬚法，Stallman 的律師告訴 CNRI 的律師，1.6.1 與 GPL「不相容」。

感謝許多的外部志工，在 Guido 指導下的付出，使得這些版本的發⬚成⬚可能。

C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和⬚明文件的授權是基於 Python 軟體基金會授權第二版 (Python Software Foundation License Version 2)。

從 Python 3.8.6 開始，⬚明文件中的範例、程式庫和其他程式碼，是被雙重授權 (dual licensed) 於 PSF 授權第二版以及 *Zero-Clause BSD 授權*。

有些被納入 Python 中的軟體是基於不同的授權。這些授權將會與其授權之程式碼一起被列出。關於這些授權的不完整清單，請參⬚被收⬚軟體的授權與致謝。

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") 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 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python 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 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.

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

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

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

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

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

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

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

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

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
```

```
    internet using the following unique, persistent identifier (known as a handle):
    1895.22/1013.  This Agreement may also be obtained from a proxy server on the
    internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

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

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

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

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTA-TION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

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

C.3 被收⬚軟體的授權與致謝

本節是一個不完整但持續增加的授權與致謝清單，對象是在 Python 發⬚版本中所收⬚的第三方軟體。

C.3.1 Mersenne Twister

`random` 模組底下的 `_random` C 擴充程式包含了以 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 的下載⬚容⬚基礎的程式碼。以下是原始程式碼的完整聲明：

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

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

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

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

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

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

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

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
```

```
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


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

C.3.2 Sockets

socket 模組使用 `getaddrinfo()` 和 `getnameinfo()` 函式，它們在 WIDE 專案（https://www.wide.ad.jp/）中，於不同的原始檔案中被編碼：

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

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

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

C.3.3 非同步 socket 服務

`test.support.asynchat` 和 `test.support.asyncore` 模組包含以下聲明：

```
Copyright 1996 by Sam Rushing

                    All Rights Reserved

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

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
```

C.3.4 Cookie 管理

`http.cookies` 模組包含以下聲明：

C.3.5 執行追⬚

`trace` 模組包含以下聲明：

```
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 與 UUdecode 函式

uu 編解碼器包含以下聲明：

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                    All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 遠端程序呼叫

xmlrpc.client 模組包含以下聲明：

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

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

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test.test_epoll` 模組包含以下聲明：

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

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

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

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

C.3.9 Select kqueue

`select` 模組對於 kqueue 介面包含以下聲明：

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

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

C.3.10 SipHash24

Python/pyhash.c 檔案包含 Marek Majkowski' 基於 Dan Bernstein 的 SipHash24 演算法的實作。它包含以下聲明:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

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

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

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 與 dtoa

Python/dtoa.c 檔案提供了 C 的 dtoa 和 strtod 函式,用於將 C 的雙精度浮點數和字串互相轉Ⓕ。該檔案是衍生自 David M. Gay 建立的同名檔案,後者現在可以從 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 下載。於 2009 年 3 月 16 日所檢索的原始檔案包含以下版權與授權聲明:

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

C.3.12 OpenSSL

如果 OpenSSL 函式庫可被作業系統使用,則 hashlib、posix、ssl 模組會使用它來提升效能。此外,因Ⓕ Windows 和 macOS 的 Python 安裝程式可能包含 OpenSSL 函式庫的副本,所以我們也在此收Ⓕ OpenSSL 授權的副本。對於 OpenSSL 3.0 版本以及由此衍生的更新版本則適用 Apache 許可證 v2:

```
                  Apache License
            Version 2.0, January 2004
         https://www.apache.org/licenses/
```

```
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.
```

```
2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

```
     for any such Derivative Works as a whole, provided Your use,
     reproduction, and distribution of the Work otherwise complies with
     the conditions stated in this License.

  5. Submission of Contributions. Unless You explicitly state otherwise,
     any Contribution intentionally submitted for inclusion in the Work
     by You to the Licensor shall be under the terms and conditions of
     this License, without any additional terms or conditions.
     Notwithstanding the above, nothing herein shall supersede or modify
     the terms of any separate license agreement you may have executed
     with Licensor regarding such Contributions.

  6. Trademarks. This License does not grant permission to use the trade
     names, trademarks, service marks, or product names of the Licensor,
     except as required for reasonable and customary use in describing the
     origin of the Work and reproducing the content of the NOTICE file.

  7. Disclaimer of Warranty. Unless required by applicable law or
     agreed to in writing, Licensor provides the Work (and each
     Contributor provides its Contributions) on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
     implied, including, without limitation, any warranties or conditions
     of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
     PARTICULAR PURPOSE. You are solely responsible for determining the
     appropriateness of using or redistributing the Work and assume any
     risks associated with Your exercise of permissions under this License.

  8. Limitation of Liability. In no event and under no legal theory,
     whether in tort (including negligence), contract, or otherwise,
     unless required by applicable law (such as deliberate and grossly
     negligent acts) or agreed to in writing, shall any Contributor be
     liable to You for damages, including any direct, indirect, special,
     incidental, or consequential damages of any character arising as a
     result of this License or out of the use or inability to use the
     Work (including but not limited to damages for loss of goodwill,
     work stoppage, computer failure or malfunction, or any and all
     other commercial damages or losses), even if such Contributor
     has been advised of the possibility of such damages.

  9. Accepting Warranty or Additional Liability. While redistributing
     the Work or Derivative Works thereof, You may choose to offer,
     and charge a fee for, acceptance of support, warranty, indemnity,
     or other liability obligations and/or rights consistent with this
     License. However, in accepting such obligations, You may act only
     on Your own behalf and on Your sole responsibility, not on behalf
     of any other Contributor, and only if You agree to indemnify,
     defend, and hold each Contributor harmless for any liability
     incurred by, or claims asserted against, such Contributor by reason
     of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

除非在建置 `pyexpat` 擴充時設定⬚ `--with-system-expat`，否則該擴充會用一個⬚含 expat 原始碼的副本來建置:

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

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

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

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

C.3.14 libffi

除非在建置 `_ctypes` 模組底下 `_ctypes` 擴充程式時設定⌴ `--with-system-libffi`，否則該擴充會用一個⌴含 libffi 原始碼的副本來建置：

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

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

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

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

C.3.15 zlib

如果在系統上找到的 zlib 版本太舊以致於無法用於建置 zlib 擴充，則該擴充會用一個⌴含 zlib 原始碼的副本來建置：

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
```

```
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.


Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

tracemalloc 使用的雜⬚表 (hash table) 實作，是以 cfuhash 專案⬚基礎：

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

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

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

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

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

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

C.3.17 libmpdec

除非在建置 decimal 模組底下 _decimal C 擴充程式時設定⬚ --with-system-libmpdec，否則該模組會用一個⬚含 libmpdec 函式庫的副本來建置：

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.
```

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

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

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

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

C.3.18 W3C C14N 測試套件

`test` 程式包中的 C14N 2.0 測試套件 (`Lib/test/xmltestdata/c14n-20/`) 是從 W3C 網站 https://www.w3.org/TR/xml-c14n2-testcases/ 被檢索，且是基於 3-clause BSD 授權被發⊞：

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

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

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

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

C.3.19 mimalloc

MIT 授權:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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

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

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

C.3.20 asyncio

asyncio 模組的部分⊞容是從 uvloop 0.16 中收⊞過來，其基於 MIT 授權來發⊞:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

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

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

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

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
```

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

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

版權宣告

Python 和這份囗明文件的版權：

完整的授權條款資訊請參見沿革與授權。

Q

R

S

X

xor
 bitwise (位元), 90

Y

yield
 expression, 81
 keyword (關鍵字), 81
 statement (陳述式), 102
 範例, 83

Z

Zen of Python (Python 之⊞), **166**
ZeroDivisionError
 exception, 89