The Python Language Reference

Δημοσίευση 3.9.23

Guido van Rossum
and the Python development team

Ιουλίου 08, 2025

Python Software Foundation
Email: docs@python.org

Περιεχόμενα

This reference manual describes the syntax and «core semantics» of the language. It is terse, but attempts to be exact and complete. The semantics of non-essential built-in object types and of the built-in functions and modules are described in library-index. For an informal introduction to the language, see tutorial-index. For C or C++ programmers, two additional manuals exist: extending-index describes the high-level picture of how to write a Python extension module, and the c-api-index describes the interfaces available to C/C++ programmers in detail.

ΚΕΦΑΛΑΙΟ 1

Εισαγωγή

Αυτό το εγχειρίδιο αναφοράς περιγράφει την γλώσσα προγραμματισμού Python. Δεν προορίζεται ως εγχειρίδιο εκμάθησης.

Στην προσπάθεια το έγγραφο αυτό να είναι όσο το δυνατόν πιο ακριβές, επιλέχθηκε αρχικά η Αγγλική γλώσσα, και ύστερα μεταφράστηκε στην Ελληνική, και όχι οι επίσημες προδιαγραφές, με εξαίρεση την συντακτική και λεξιλογική ανάλυση. Αυτό θα πρέπει να κάνει το έγγραφο πιο κατανοητό στον μέσο αναγνώστη, αλλά θα αφήσει χώρο για αμφισημίες. Συνεπώς, αν ερχόσουν από τον Άρη και προσπαθούσες να υλοποιήσεις ξανά την Python από το έγγραφο αυτό και μόνο, μάλλον θα χρειαζόταν να μαντέψεις κάποια πράγματα και για την ακρίβεια ίσως θα κατέληγες να υλοποιείς μια τελείως διαφορετική γλώσσα. Από την άλλη πλευρά, αν χρησιμοποιείς την Python και αναρωτιέσαι ποιοι είναι οι ακριβείς κανόνες σχετικά με έναν συγκεκριμένο τομέα της γλώσσας, τότε σίγουρα θα τους βρεις εδώ πέρα. Αν θα ήθελες να δεις έναν πιο επίσημο ορισμό της γλώσσας, ίσως θα μπορούσες να προσφέρεις λίγο από τον χρόνο σου — ή να φτιάξεις μια μηχανή κλωνοποίησης :-).

Είναι επικίνδυνο να προσθέσουμε πολλές λεπτομέρειες υλοποίησης σε ένα έγγραφο αναφοράς μίας γλώσσας — η υλοποίηση δύναται να αλλάξει, και άλλες υλοποιήσεις της ίδιας γλώσσας μπορεί να λειτουργούν διαφορετικά. Από την άλλη, η CPython είναι μία υλοποίηση της Python με ευρεία χρήση (ωστόσο εναλλακτικές υλοποιήσεις συνεχίζουν να υποστηρίζονται), και οι συγκεκριμένες της ιδιομορφίες ενίοτε αξίζουν αναφορά, ειδικά εκεί που η υλοποίηση επιβάλλει επιπρόσθετους περιορισμούς. Επομένως, θα βρεις σύντομες «σημειώσεις υλοποίησης» σε διάφορα μέρη του κειμένου.

Κάθε υλοποίηση της Python συνοδεύεται από έναν αριθμό ενσωματωμένων και πρότυπων module. Αυτές είναι καταγεγραμμένες στο library-index. Κάποια ενσωματωμένα module αναφέρονται όταν αλληλεπιδρούν με έναν σημαντικό τρόπο με τον ορισμό της γλώσσας.

1.1 Εναλλακτικές Υλοποιήσεις

Παρόλο που υπάρχει μία υλοποίηση της Python που είναι μακράν η πιο διάσημη, υπάρχουν εναλλακτικές υλοποιήσεις που έχουν ιδιαίτερο ενδιαφέρον για διάφορους ανθρώπους.

Γνωστές υλοποιήσεις περιλαμβάνουν:

CPython Αυτή είναι η πρωτότυπη και η πιο καλοδιατηρημένη υλοποίηση της Python, γραμμένη στην C. Νέες λειτουργίες της γλώσσας συνήθως εμφανίζονται πρώτα εδώ.

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 για το .NET Αυτή η υλοποίηση στην πραγματικότητα χρησιμοποιεί την υλοποίηση CPython, αλλά είναι μία διαχειριζόμενη εφαρμογή του .NET και κάνει διαθέσιμες τις .NET βιβλιοθήκες. Δημιουργήθηκε από τον *Brian Lloyd*. Για περισσότερες πληροφορίες, δείτε την αρχική σελίδα της Python για το .NET.

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.

Κάθε μία από αυτές τις υλοποιήσεις διαφοροποιούνται με κάποιον τρόπο από την γλώσσα όπως καταγράφεται σε αυτό το εγχειρίδιο, ή εισάγει συγκεκριμένη πληροφορία πέρα από ό,τι καλύπτουν τα πρότυπα έγγραφα της Python. Παρακαλώ να συμβουλευτείτε το έγγραφο της συγκεκριμένης υλοποίησης για να προσδιορίσετε τι άλλο χρειάζεται να ξέρετε σχετικά με την συγκεκριμένη υλοποίηση που χρησιμοποιείτε.

1.2 Σημειογραφία

The descriptions of lexical analysis and syntax use a modified BNF grammar notation. This uses the following style of definition:

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

Η πρώτη γραμμή λέει ότι ένα `name` είναι ένα `lc_letter` ακολουθούμενο από μία σειρά από μηδέν ή περισσότερα `lc_letters` και κάτω παύλες. Ένα `lc_letter` με τη σειρά του είναι οποιοσδήποτε από τους μονούς χαρακτήρες `'a'` έως `'z'`. (Αυτός ο κανόνας στην πραγματικότητα εφαρμόζεται για τα ονόματα που ορίζονται στους λεξιλογικούς και γραμματικούς κανόνες αυτού του εγγράφου.)

Κάθε κανόνας ξεκινά με ένα όνομα (το οποίο είναι ένα όνομα ορισμένο από τον κανόνα) και `::=`. Μία κάθετη γραμμή (`|`) χρησιμοποιείται για να διαχωρίσει τις εναλλακτικές· έχει την μικρότερη προτεραιότητα στην σειρά προτεραιότητας πράξεων αυτού του συμβολισμού. Ένας αστερίσκος (`*`) σημαίνει μηδέν ή περισσότερες επαναλήψεις του προηγούμενου αντικειμένου· παρομοίως, το συν (`+`) σημαίνει μία ή περισσότερες επαναλήψεις, και μία φράση περιφραγμένη από αγκύλες (`[ ]`) σημαίνει μηδέν ή μία περίπτωση (με άλλα λόγια, η περιφραγμένη φράση είναι προαιρετική). Οι τελεστές `*` και `+` ενώνονται όσο το δυνατόν πιο σφιχτά· οι παρενθέσεις χρησιμοποιούνται για ομαδοποίηση. Οι συμβολοσειρές είναι περιφραγμένες από εισαγωγικά. Οι κενοί χαρακτήρες είναι μόνο σημαντικοί για να διαχωρίσουν τα *tokens*. Οι κανόνες συνήθως περιέχονται σε μία μονή γραμμή· οι κανόνες με πολλές εναλλακτικές μπορεί να μορφοποιηθούν εναλλακτικά με κάθε γραμμή μετά την πρώτη να ξεκινάει με μια κάθετη γραμμή.

Στους λεξιλογικούς ορισμούς (όπως στο παραπάνω παράδειγμα), δύο περισσότεροι κανόνες χρησιμοποιούνται: Δύο χαρακτήρες χωρισμένοι από τρεις τελείες σημαίνει επιλογή όποιου μονού χαρακτήρα στο συγκεκριμένο (κλειστό) εύρος *ASCII* χαρακτήρων. Η φράση ανάμεσα σε γωνιακές παρενθέσεις (<...>) δίνει μία άτυπη περιγραφή του ορισμένου συμβόλου· π.χ., αυτό θα μπορούσε να χρησιμοποιηθεί για να περιγράψει την ιδέα του "χαρακτήρα ελέγχου" (control character) αν χρειαστεί.

Αν και η σημειογραφία που χρησιμοποιείται είναι σχεδόν η ίδια, υπάρχει μεγάλη διαφορά ανάμεσα στη σημασία των λεξιλογικών και των συντακτικών ορισμών: ένας λεξιλογικός ορισμός λειτουργεί με τους μεμονωμένους χαρακτήρες της πηγής εισόδου, ενώ ένας ορισμός σύνταξης λειτουργεί στην ροή των *token* που δημιουργείται από τη λεξιλογική ανάλυση. Όλες οι χρήσεις του *BNF* στο επόμενο κεφάλαιο («Λεξιλογική Ανάλυση») είναι λεξιλογικοί ορισμοί· οι χρήσεις στα ακόλουθα κεφάλαια είναι συντακτικοί ορισμοί.

ΚΕΦΑΛΑΙΟ 2

Lexical analysis

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. In addition, if the first bytes of the file are the UTF-8 byte-order mark (`b'\xef\xbb\xbf'`), the declared file encoding is UTF-8 (this is supported, among others, by Microsoft's **notepad**).

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:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Looks like a valid date
        return 1
```

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:

```
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:

```
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:

```
 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 are the same as in Python 2.x: the uppercase and lowercase letters `A` through `Z`, the underscore _ and, except for the first character, the digits `0` through `9`.

Python 3.0 introduces 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 under
id_continue  ::=  <all characters in id_start, plus characters in the categories Mn, Mc,
xid_start    ::=  <all characters in id_start whose NFKC normalization is in "id_start xi
xid_continue ::=  <all characters in id_continue whose NFKC normalization is in "id_conti
```

The Unicode category codes mentioned above stand for:

- *Lu* - uppercase letters
- *Ll* - lowercase letters
- *Lt* - titlecase letters

- *Lm* - modifier letters
- *Lo* - other letters
- *Nl* - letter numbers
- *Mn* - nonspacing marks
- *Mc* - spacing combining marks
- *Nd* - decimal numbers
- *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 4.1 can be found at https://www.unicode.org/Public/13.0.0/ucd/DerivedCoreProperties.txt

2.3.1 Keywords

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 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 *`. The special identifier _ is used in the interactive interpreter to store the result of the last evaluation; it is stored in the `builtins` module. When not in interactive mode, _ has no special meaning and is not defined. See section *The import statement*.

> **Σημείωση:** The name _ is often used in conjunction with internationalization; refer to the documentation for the `gettext` module for more information on this convention.

__*__ 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 strings are called *raw strings* and treat backslashes as literal characters. As a result, in string literals, `'\U'` and `'\u'` escapes in raw strings are not treated specially. Given that Python 2.x's raw unicode literals behave differently than Python 3.x's the `'ur'` syntax is not supported.

Νέο στην έκδοση 3.3: The `'rb'` prefix of raw bytes literals has been added as a synonym of `'br'`.

Νέο στην έκδοση 3.3: 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 *Formatted string literals*. 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 `"`.)

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	Meaning	Notes
`\newline`	Backslash and newline ignored	
`\\`	Backslash (`\`)	
`\'`	Single quote (`'`)	
`\"`	Double quote (`"`)	
`\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*	(1,3)
`\xhh`	Character with hex value *hh*	(2,3)

Escape sequences only recognized in string literals are:

Escape Sequence	Meaning	Notes
`\N{name}`	Character named *name* in the Unicode database	(4)
`\uxxxx`	Character with 16-bit hex value *xxxx*	(5)
`\Uxxxxxxxx`	Character with 32-bit hex value *xxxxxxxx*	(6)

Notes:

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

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

(3) 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.

(4) Άλλαξε στην έκδοση 3.3: Support for name aliases[1] has been added.

(5) Exactly four hex digits are required.

(6) 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.

Άλλαξε στην έκδοση 3.6: Unrecognized escape sequences produce a `DeprecationWarning`. In a future Python version they will be a `SyntaxWarning` and eventually a `SyntaxError`.

[1] https://www.unicode.org/Public/11.0.0/ucd/NameAliases.txt

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 Formatted string literals

Νέο στην έκδοση 3.6.

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 | NULL | 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. Replacement expressions can contain line breaks (e.g. in triple-quoted strings), but they cannot contain comments. Each expression is evaluated in the context where the formatted string literal appears, in order from left to right.

Άλλαξε στην έκδοση 3.7: 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.

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.

Νέο στην έκδοση 3.8: The equal sign `'='`.

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" '
```

A consequence of sharing the same syntax as regular string literals is that characters in the replacement fields must not conflict with the quoting used in the outer formatted string literal:

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

Backslashes are not allowed in format expressions and will raise an error:

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

To include a value in which a backslash escape is required, create a temporary variable.

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

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
```

Άλλαξε στην έκδοση 3.6: Underscores are now allowed for grouping purposes in literals.

2.4.6 Floating point literals

Floating point literals are described by the following lexical definitions:

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

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

Some examples of floating point literals:

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

Άλλαξε στην έκδοση 3.6: 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:

```
$       ?       `
```

ΚΕΦΑΛΑΙΟ 3

Data model

3.1 Objects, values and types

Objects are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. (In a sense, and in conformance to Von Neumann's model of a «stored program computer», code is also represented by objects.)

Every object has an identity, a type and a value. An object's *identity* never changes once it has been created; you may think of it as the object's address in memory. The "`is`" operator compares the identity of two objects; the `id()` function returns an integer representing its identity.

CPython implementation detail: For CPython, `id(x)` is the memory address where `x` is stored.

An object's type determines the operations that the object supports (e.g., «does it have a length?») and also defines the possible values for objects of that type. The `type()` function returns an object's type (which is an object itself). Like its identity, an object's *type* is also unchangeable.[1]

The *value* of some objects can change. Objects whose value can change are said to be *mutable*; objects whose value is unchangeable once they are created are called *immutable*. (The value of an immutable container object that contains a reference to a mutable object can change when the latter's value is changed; however the container is still considered immutable, because the collection of objects it contains cannot be changed. So, immutability is not strictly the same as having an unchangeable value, it is more subtle.) An object's mutability is determined by its type; for instance, numbers, strings and tuples are immutable, while dictionaries and lists are mutable.

Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected. An implementation is allowed to postpone garbage collection or omit it altogether — it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.

CPython implementation detail: CPython currently uses a reference-counting scheme with (optional) delayed detection of cyclically linked garbage, which collects most objects as soon as they become unreachable, but is not guaranteed to collect garbage containing circular references. See the documentation of the `gc` module for information on controlling the collection of cyclic garbage. Other implementations act differently and CPython may change. Do not depend on immediate finalization of objects when they become unreachable (so you should always close files explicitly).

[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.

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

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

Some objects contain references to other objects; these are called *containers*. Examples of containers are tuples, lists and dictionaries. The references are part of a container's value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the mutability of a container, only the identities of the immediately contained objects are implied. So, if an immutable container (like a tuple) contains a reference to a mutable object, its value changes if that mutable object is changed.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. E.g., after `a = 1; b = 1`, `a` and `b` may or may not refer to the same object with the value one, depending on the implementation, but after `c = []; d = []`, `c` and `d` are guaranteed to refer to two different, unique, newly created empty lists. (Note that `c = d = []` assigns the same object to both `c` and `d`.)

3.2 The standard type hierarchy

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.

None This type has a single value. There is a single object with this value. This object is accessed through the built-in name `None`. It is used to signify the absence of a value in many situations, e.g., it is returned from functions that don't explicitly return anything. Its truth value is false.

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

See implementing-the-arithmetic-operations for more details.

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

Ellipsis This type has a single value. There is a single object with this value. This object is accessed through the literal `...` or the built-in name `Ellipsis`. Its truth value is true.

`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).

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.

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

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`.

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]`.

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. This implies that the index set is renumbered so that it starts at 0.

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:

Strings 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.

Tuples 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.

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

Mutable sequences 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.

There are currently two intrinsic mutable sequence types:

Lists 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.)

Byte Arrays 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.

The extension module `array` provides an additional example of a mutable sequence type, as does the `collections` module.

Set types 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:

Sets 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 sets 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.

Mappings 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:

Dictionaries 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.

Άλλαξε στην έκδοση 3.7: 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.

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

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

Special attributes:

Attribute	Meaning	
`__doc__`	The function's documentation string, or `None` if unavailable; not inherited by subclasses.	Writable
`__name__`	The function's name.	Writable
`__qualname__`	The function's *qualified name*. Νέο στην έκδοση 3.3.	Writable
`__module__`	The name of the module the function was defined in, or `None` if unavailable.	Writable
`__defaults__`	A tuple containing default argument values for those arguments that have defaults, or `None` if no arguments have a default value.	Writable
`__code__`	The code object representing the compiled function body.	Writable
`__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.	Read-only
`__dict__`	The namespace supporting arbitrary function attributes.	Writable
`__closure__`	`None` or a tuple of cells that contain bindings for the function's free variables. See below for information on the `cell_contents` attribute.	Read-only
`__annotations__`	A dict containing annotations of parameters. The keys of the dict are the parameter names, and `'return'` for the return annotation, if provided.	Writable
`__kwdefaults__`	A dict containing defaults for keyword-only parameters.	Writable

Most of the attributes labelled «Writable» check the type of the assigned value.

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. *Note that the current implementation only supports function attributes on user-defined functions. Function attributes on built-in functions may be supported in the future.*

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.

Additional information about a function's definition can be retrieved from its code object; see the description of internal types below. The `cell` type can be accessed in the `types` module.

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

Special read-only attributes: `__self__` is the class instance object, `__func__` is the function object; `__doc__` is the method's documentation (same as `__func__.__doc__`); `__name__` is the method name (same as `__func__.__name__`); `__module__` is 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 class method 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 class method 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 class method object, the «class instance» stored in `__self__` will actually be the class itself, so that calling either `x.f(1)` or `C.f(1)` is equivalent to calling `f(C,1)` where `f` is the underlying function.

Note that the transformation from function object to instance method object happens each time the attribute is retrieved from the instance. In some cases, a fruitful optimization is to assign the attribute to a local variable and call that local variable. Also notice that this transformation only happens for user-defined functions; other callable objects (and all non-callable objects) are retrieved without transformation. It is also important to note that user-defined functions which are attributes of a class instance are not converted to bound methods; this *only* happens when the function is an attribute of the class.

Generator functions A function or method which uses the `yield` statement (see section *The yield statement*) 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 functions 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 *Coroutine Objects* section.

Asynchronous generator functions 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.

Built-in functions A built-in function object is a wrapper around a C function. Examples of built-in functions are `len()` and `math.sin()` (`math` is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes: `__doc__` is the function's documentation string, or `None` if unavailable; `__name__` is the function's 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.

Built-in methods 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*.

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.

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

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

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

Predefined (writable) attributes: `__name__` is the module's name; `__doc__` is the module's documentation string, or `None` if unavailable; `__annotations__` (optional) is a dictionary containing *variable annotations* collected during module body execution; `__file__` is the pathname of the file from which the module was loaded, if it was loaded from a file. The `__file__` attribute may be missing for certain types of modules, such as C modules that are statically linked into the interpreter; for extension modules loaded dynamically from a shared library, it is the pathname of the shared library file.

Special read-only attribute: `__dict__` is the module's namespace as a dictionary object.

CPython implementation detail: 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.

Custom classes Custom class types are typically created by class definitions (see section *Class definitions*). 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 in the documentation accompanying the 2.3 release at https://www.python.org/download/releases/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 static method object, it is transformed into the object wrapped by the static method object. See section *Implementing Descriptors* 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).

Special attributes: `__name__` is the class name; `__module__` is the module name in which the class was defined; `__dict__` is the dictionary containing the class's namespace; `__bases__` is a tuple containing the base classes, in the order of their occurrence in the base class list; `__doc__` is the class's documentation string, or `None` if undefined; `__annotations__` (optional) is a dictionary containing *variable annotations* collected during class body execution.

Class instances 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 *Implementing Descriptors* 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*.

Special attributes: `__dict__` is the attribute dictionary; `__class__` is the instance's class.

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.

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.

Code objects 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.

Special read-only attributes: `co_name` gives the function name; `co_argcount` is the total number of positional arguments (including positional-only arguments and arguments with default values); `co_posonlyargcount` is the number of positional-only arguments (including arguments with default values); `co_kwonlyargcount` is the number of keyword-only arguments (including arguments with default values); `co_nlocals` is the number of local variables used by the function (including arguments); `co_varnames` is a tuple containing the names of the local variables (starting with the argument names); `co_cellvars` is a tuple containing the names of local variables that are referenced by nested functions; `co_freevars` is a tuple containing the names of free variables; `co_code` is a string representing the sequence of bytecode instructions; `co_consts` is a tuple containing the literals used by the bytecode; `co_names` is a tuple containing the names used by the bytecode; `co_filename` is the filename from which the code was compiled; `co_firstlineno` is the first line number of the function; `co_lnotab` is a string encoding the mapping from bytecode offsets to line numbers (for details see the source code of the interpreter); `co_stacksize` is the required stack size; `co_flags` is 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.

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.

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

Special read-only attributes: `f_back` is to the previous stack frame (towards the caller), or `None` if this is the bottom stack frame; `f_code` is the code object being executed in this frame; `f_locals` is the dictionary used to look up local variables; `f_globals` is used for global variables; `f_builtins` is used for built-in (intrinsic) names; `f_lasti` gives the precise instruction (this is an index into the bytecode string of the code object).

Accessing `f_code` raises an auditing event `object.__getattr__` with arguments `obj` and `"f_code"`.

Special writable attributes: `f_trace`, if not `None`, is a function called for various events during code execution (this is used by the debugger). Normally an event is triggered for each new source line - this can be disabled by setting `f_trace_lines` to `False`.

Implementations *may* allow per-opcode events to be requested by setting `f_trace_opcodes` to `True`. Note that this may lead to undefined interpreter behaviour if exceptions raised by the trace function escape to the function being traced.

`f_lineno` is 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 f_lineno.

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.

Νέο στην έκδοση 3.4.

Traceback objects Traceback objects represent a 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`.

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 *The try statement*.) It is accessible as the third item of the tuple returned by `sys.exc_info()`, and as the `__traceback__` attribute of the caught exception.

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

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

Special read-only attributes: `tb_frame` points to the execution frame of the current level; `tb_lineno` gives the line number where the exception occurred; `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.

Accessing `tb_frame` raises an auditing event `object.__getattr__` with arguments `obj` and `"tb_frame"`.

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.

Άλλαξε στην έκδοση 3.7: Traceback objects can now be explicitly instantiated from Python code, and the `tb_next` attribute of existing instances can be updated.

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 not themselves callable, although the objects they wrap usually are. 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 «User-defined 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.)

[2] The `__hash__()`, `__iter__()`, `__reversed__()`, and `__contains__()` 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.

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.

Σημείωση: `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.

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

Δείτε επίσης:

Documentation for the `gc` module.

> **Προειδοποίηση:** 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.

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

Called by `str(object)` and the built-in functions `format()` and `print()` to compute the «informal» or nicely printable string representation of an object. The return value must be a string 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.

`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.

The return value must be a string object.

Άλλαξε στην έκδοση 3.4: The __format__ method of `object` itself raises a `TypeError` if passed any non-empty string.

Άλλαξε στην έκδοση 3.7: `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()`.

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 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.

`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:

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

Σημείωση: `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; 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.

Σημείωση: 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://www.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).

See also `PYTHONHASHSEED`.

Άλλαξε στην έκδοση 3.3: 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__()*, 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.

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 fake 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)`.

Σημείωση: 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. A sequence must be returned. `dir()` converts the returned sequence 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 a sequence 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:

```
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
```

Σημείωση: 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.

Άλλαξε στην έκδοση 3.5: `__class__` module attribute is now writable.

Νέο στην έκδοση 3.7: `__getattr__` and `__dir__` module attributes.

Δείτε επίσης:

PEP 562 - Module __getattr__ and __dir__ Describes the `__getattr__` and `__dir__` functions on modules.

Implementing Descriptors

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__`.

`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.

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

Called at the time the owning class *owner* is created. The descriptor has been assigned to *name*.

Σημείωση: `__set_name__()` is only called implicitly as part of the `type` constructor, so it will need to be called explicitly with the appropriate parameters when a descriptor is added to a class after initial creation:

```
class A:
   pass
descr = custom_descriptor()
A.attr = descr
descr.__set_name__(A, 'attr')
```

See *Creating the class object* for more details.

Νέο στην έκδοση 3.6.

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`:

Direct Call 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 If `a` is an instance of `super`, then the binding `super(B, obj).m()` searches `obj.__class__.__mro__` for the base class `A` immediately following `B` and then invokes the descriptor with the call: `A.__dict__['m'].__get__(obj, obj.__class__)`.

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, child subclasses will get a `__dict__` and *__weakref__* unless they also define *__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.
- Nonempty *__slots__* does not work for classes derived from «variable-length» built-in types 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's class `__init_subclass__`. For compatibility with other classes using `__init_subclass__`, one should take out the needed keyword arguments and pass the others over to the base class, as in:

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

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

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

Σημείωση: 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 accessed as `type(cls)`.

Νέο στην έκδοση 3.6.

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

If a base that appears in class definition is not an instance of `type`, then an `__mro_entries__` method is searched on it. If found, it is called with the original bases tuple. This method must return a tuple of classes that will be used instead of this base. The tuple may be empty, in such case the original base is ignored.

Δείτε επίσης:

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.

Δείτε επίσης:

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 implementation detail: 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 customisation steps are invoked after creating the class object:

- first, `type.__new__` collects all of the descriptors in the class namespace that define a *`__set_name__()`* method;
- second, all of these `__set_name__` methods are called with the class being defined and the assigned name of that particular descriptor;
- finally, 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.

Δείτε επίσης:

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.

`class.`**`__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)`.

`class.`**`__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.

Δείτε επίσης:

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`.

Δείτε επίσης:

PEP 484 - Type Hints Introducing Python's framework for type annotations

Generic Alias Types 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:

```
from inspect import isclass

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

    class_of_obj = type(obj)

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

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

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

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

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

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

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

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
>>> type(Menu['SPAM'])
<enum 'Menu'>
```

Δείτε επίσης:

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, ...)`.

3.3.7 Emulating container types

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

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

CPython implementation detail: 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.

Νέο στην έκδοση 3.4.

Σημείωση: 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 and slice objects. Note that the special interpretation of negative indexes (if the class wishes to emulate a *sequence* type) is up to the *`__getitem__()`* method. If *key* is of an inappropriate type, `TypeError` may be raised; if of 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.

Σημείωση: *`for`* loops expect that an `IndexError` will be raised for illegal indexes to allow proper detection of the end of the sequence.

Σημείωση: 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.

Iterator objects also need to implement this method; they are required to return themselves. For more information on iterator objects, see typeiter.

`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, `x.__add__(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, `y.__rsub__(x)` is called if `x.__sub__(y)` returns *NotImplemented*.

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

Σημείωση: 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, 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)` . Otherwise, `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.

Σημείωση: Due to a bug in the dispatching mechanism for `**=`, a class that defines `__ipow__()` but returns `NotImplemented` would fail to fall back to `x.__pow__(y)` and `y.__rpow__(x)`. This bug is fixed in Python 3.10.

`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.

3.3.9 With Statement Context Managers

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 *The with statement*), 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.

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.

Δείτε επίσης:

PEP 343 - The «with» statement The specification, background, and examples for the Python *with* statement.

3.3.10 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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
>>> 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 Coroutines

3.4.1 Awaitable Objects

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

Σημείωση: The *generator iterator* objects returned from generators decorated with `types.coroutine()` or `asyncio.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.

Νέο στην έκδοση 3.5.

Δείτε επίσης:

PEP 492 for additional information about awaitable objects.

3.4.2 Coroutine Objects

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.

Άλλαξε στην έκδοση 3.5.2: 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.

`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.

`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:

```
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
```

Νέο στην έκδοση 3.5.

Άλλαξε στην έκδοση 3.7: 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.

`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:

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

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

Νέο στην έκδοση 3.5.

ΚΕΦΑΛΑΙΟ 4

Μοντέλο εκτέλεσης

4.1 Δομή ενός προγράμματος

Ένα πρόγραμμα Python αποτελείται από μπλοκ κώδικα. Ένα *μπλοκ* είναι ένα κομμάτι κειμένου προγράμματος Python που εκτελείται ως μια μονάδα. Τα παρακάτω είναι μπλοκ: ένα module, το σώμα μιας συνάρτησης, ο ένας ορισμός κλάσης. Κάθε εντολή που πληκτρολογείται διαδραστικά αποτελεί μπλοκ. Ένα αρχείο δέσμης ενεργειών (ένα αρχείο που δίνεται ως τυπική είσοδος στο διερμηνέα ή καθορίζεται ως όρισμα γραμμής εντολών στον διερμηνέα) είναι ένα μπλοκ κώδικα. Μια script εντολή (μια εντολή που καθορίζεται στο διερμηνέα με την επιλογή `-c`) είναι ένα μπλοκ κώδικα. Μια ενότητα που εκτελείται ως ανωτέρου επιπέδου script (ως module `__main__`) από τη γραμμή εντολών χρησιμοποιώντας ένα όρισμα `-m` όρισμα είναι επίσης ένα μπλοκ κώδικα. Το όρισμα συμβολοσειράς που περνάει στις ενσωματωμένες συναρτήσεις `eval()` και `exec()` είναι ένα μπλοκ κώδικα.

Ένα μπλοκ κώδικα εκτελείται σε ένα *πλαίσιο εκτέλεσης*. Ένα πλαίσιο περιέχει ορισμένες πληροφορίες διαχείρισης (που χρησιμοποιούνται για αποσφαλμάτωση) και καθορίζει που και πως συνεχίζεται η εκτέλεση μετά την ολοκλήρωση της εκτέλεσης του μπλοκ κώδικα.

4.2 Ονομασία και σύνδεση

4.2.1 Σύνδεση ονομάτων

Names αναφέρονται σε αντικείμενα. Τα ονόματα εισάγονται μέσω λειτουργιών δέσμευσης ονομάτων.

The following constructs bind names: formal parameters to functions, *import* statements, class and function definitions (these bind the class or function name in the defining block), and targets that are identifiers if occurring in an assignment, *for* loop header, or after `as` in a *with* statement or *except* clause. 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.

Ένας στόχος που εμφανίζεται σε μια δήλωση *del* θεωρείται επίσης δεσμευμένος για αυτό τον σκοπό (αν και η πραγματική σημασιολογία είναι να αποσυνδέσει το όνομα).

Κάθε δήλωση ανάθεσης ή εισαγωγής συμβαίνει μέσα σε ένα μπλοκ που ορίζεται από έναν ορισμό κλάσης ή συνάρτησης ή στο επίπεδο του module (το μπλοκ κώδικα ανώτατου επιπέδου).

Αν ένα όνομα δεσμεύεται σε ένα μπλοκ, είναι μια τοπική μεταβλητή αυτού του μπλοκ, εκτός αν δηλωθεί ως `nonlocal` ή `global`. Αν ένα όνομα δεσμεύεται στο επίπεδο του module, είναι μια καθολική μεταβλητή. (Οι μεταβλητές του μπλοκ του module είναι ταυτόχρονα τοπικές και καθολικές.) Αν μια μεταβλητή χρησιμοποιείται σε ένα μπλοκ κώδικα αλλά δεν ορίζεται εκεί, είναι μια *free variable*.

Κάθε εμφάνιση ενός ονόματος στο κείμενο του προγράμματος αναφέρεται στη *binding* αυτού του ονόματος που καθορίζεται από τους παρακάτω κανόνες επίλυσης ονομάτων.

4.2.2 Επίλυση ονομάτων

Ένα *scope* ορίζει την ορατότητα ενός ονόματος μέσα σε ένα μπλοκ. Αν μια τοπική μεταβλητή οριστεί σε ένα μπλοκ, το πεδίο της περιλαμβάνει το μπλοκ αυτό. Αν ο ορισμός συμβαίνει σε ένα μπλοκ συνάρτησης, το πεδίο επεκτείνεται σε οποιαδήποτε μπλοκ περιέχονται μέσα σε αυτό που την ορίζει, εκτός αν ένα περιεχόμενο μπλοκ εισάγει διαφορετική σύνδεση για το όνομα.

Όταν ένα όνομα χρησιμοποιείται σε ένα μπλοκ κώδικα, επιλύεται χρησιμοποιώντας το πλησιέστερο περιβάλλον πεδίο. Το σύνολο όλων των πεδίων που είναι ορατά σε ένα μπλοκ κώδικα ονομάζεται *environment* του μπλοκ.

Όταν ένα όνομα δεν βρίσκεται καθόλου, γίνεται raise μια εξαίρεση `NameError`. Αν το τρέχον πεδίο είναι πεδίο συνάρτησης και το όνομα αναφέρεται σε μια τοπική μεταβλητή που δεν έχει ακόμα δεσμευτεί σε κάποια τιμή στο σημείο που χρησιμοποιείται το όνομα, γίνεται raise μια εξαίρεση `UnboundLocalError`. Η `UnboundLocalError` είναι μια υποκλάση της `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.

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. The `global` statement must precede all uses of the listed names.

Η δήλωση `global` έχει το ίδιο πεδίο με μια λειτουργία σύνδεσης ονόματος στο ίδιο μπλοκ. Αν το πλησιέστερο περιβάλλον πεδίου για μια ελεύθερη μεταβλητή περιέχει μια δήλωση global, η ελεύθερη μεταβλητή αντιμετωπίζεται ως καθολική.

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.

Ο χώρος ονομάτων για ένα module δημιουργείται αυτόματα την πρώτη φορά που το module εισάγεται Το κύριο module για ένα script ονομάζεται πάντα `__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 since they are implemented using a function scope. This means that the following will fail:

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

4.2.3 Ενσωματωμένες συναρτήσεις και περιορισμένη εκτέλεση

CPython implementation detail: Οι χρήστες δεν θα πρέπει να τροποποιούν το `__builtins__`· είναι αυστηρά μια λεπτομέρεια υλοποίησης. Οι χρήστες που θέλουν να παρακάμψουν τιμές στον χώρο ονομάτων των ενσωματωμένων συναρτήσεων θα πρέπει να κάνουν *import* το module `builtins` και να τροποποιούν τα χαρακτηριστικά του κατάλληλα.

Ο χώρος ονομάτων των ενσωματωμένων συναρτήσεων που σχετίζεται με την εκτέλεση ενός μπλοκ κώδικα βρίσκεται στην πραγματικότητα μέσω αναζήτησης του ονόματος `__builtins__` στον καθολικό του χώρο ονομάτων· αυτό θα πρέπει να είναι ένα λεξικό ή ένα module (στη δεύτερη περίπτωση χρησιμοποιείται το λεξικό του module). Από προεπιλογή, όταν βρισκόμαστε στο module `__main__`, το `__builtins__` είναι το ενσωματωμένο module `builtins`· όταν βρισκόμαστε σε οποιοδήποτε άλλο module, το `__builtins__` είναι ένα ψευδώνυμο για το λεξικό του ίδιου του module `builtins`.

4.2.4 Αλληλεπίδραση με δυναμικές λειτουργίες

Η επίλυση ονομάτων των ελεύθερων μεταβλητών συμβαίνει κατά το χρόνο εκτέλεσης, όχι κατά το χρόνο μεταγλώττισης. Αυτό σημαίνει ότι ο παρακάτω κώδικας θα εκτυπώσει το 42:

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

Οι συναρτήσεις `eval()` και `exec()` δεν έχουν πρόσβαση στο πλήρες περιβάλλον για την επίλυση ονομάτων. Τα ονόματα μπορεί να επιλύονται στους τοπικούς και καθολικούς χώρους ονομάτων του καλούντος. Οι ελεύθερες μεταβλητές δεν επιλύονται στο πλησιέστερο περιβάλλον πεδίου, αλλά στον καθολικό χώρο ονομάτων.[1] Οι συναρτήσεις `exec()` και `eval()` έχουν προαιρετικά ορίσματα για να παρακάμψουν τους καθολικούς και τοπικούς χώρους ονομάτων. Αν καθοριστεί μόνο ένας χώρος ονομάτων, χρησιμοποιείται και για τους δύο.

4.3 Εξαιρέσεις

Οι εξαιρέσεις είναι ένας τρόπος διακοπής της κανονικής ροής ελέγχου ενός μπλοκ κώδικα, προκειμένου να αντιμετωπιστούν σφάλματα ή άλλες εξαιρετικές συνθήκες. Μια εξαίρεση *γίνεται raise* στο σημείο όπου εντοπίζεται το σφάλμα· μπορεί να *αντιμετωπιστεί* από το περιβάλλον μπλοκ κώδικα ή από οποιοδήποτε μπλοκ κώδικα που άμεσα ή έμμεσα εκτέλεσε το μπλοκ κώδικα όπου συνέβη το σφάλμα.

Ο διερμηνέας της Python εγείρει μια εξαίρεση όταν εντοπίσει ένα σφάλμα κατά την εκτέλεση(όπως η διαίρεση με το μηδέν). Ένα πρόγραμμα Python μπορεί επίσης να εγείρει ρητά μια εξαίρεση με τη δήλωση *raise*. Οι διαχειριστές εξαιρέσεων καθορίζονται με τη δήλωση *try* … *except*. Η ρήτρα *finally* μιας τέτοιας δήλωσης μπορεί να χρησιμοποιηθεί για να καθοριστεί κώδικας καθαρισμού, ο οποίος δεν διαχειρίζεται την εξαίρεση αλλά εκτελείται ανεξάρτητα από το αν προηγήθηκε εξαίρεση ή όχι στον προηγούμενο κώδικα.

Η Python χρησιμοποιεί το μοντέλο διαχείρισης σφαλμάτων «τερματισμού»: ένας διαχειριστής εξαιρέσεων μπορεί να διαπιστώσει τι συνέβη και να συνεχίσει την εκτέλεση σε ένα εξωτερικό επίπεδο, αλλά δεν μπορεί να διορθώσει την αιτία του σφάλματος και να επαναλάβει τη λειτουργία που απέτυχε (εκτός αν επανεισαχθεί το προβληματικό κομμάτι κώδικα από την αρχή).

Όταν μια εξαίρεση δεν αντιμετωπιστεί καθόλου, ο διερμηνέας τερματίζει την εκτέλεση του προγράμματος ή επιστρέφει στον διαδραστικό κύριο βρόχο του. Και στις δύο περιπτώσεις, εκτυπώνει το ίχνος της στοίβας, εκτός αν η εξαίρεση είναι `SystemExit`.

[1] Αυτός ο περιορισμός προκύπτει επειδή ο κώδικας που εκτελείται από αυτές τις λειτουργίες δεν είναι διαθέσιμος τη στιγμή που το module μεταγλωττίζεται.

Οι εξαιρέσεις αναγνωρίζονται από στιγμιότυπα κλάσεων. Η ρήτρα `except` επιλέγεται ανάλογα με την κλάση του στιγμιότυπου: πρέπει να αναφέρεται στην κλάση του στιγμιότυπου ή σε μια *μη εικονική βασική κλάση* αυτής. Το στιγμιότυπο μπορεί να παραληφθεί από τον διαχειριστή και να μεταφέρει πρόσθετες πληροφορίες σχετικά με την εξαιρετική συνθήκη.

Σημείωση: Τα μηνύματα εξαιρέσεων δεν αποτελούν μέρος του API της Python. Το περιεχόμενό τους μπορεί να αλλάξει από τη μία έκδοση της Python στην επόμενη χωρίς προειδοποίηση και δεν θα πρέπει να βασίζεται σε αυτά ο κώδικας που θα εκτελεστεί σε πολλαπλές εκδόσεις του διερμηνέα.

Δείτε επίσης την περιγραφή της δήλωσης `try` στην ενότητα *The try statement* και της δήλωσης `raise` στην ενότητα *The raise statement*.

Υποσημειώσεις

ΚΕΦΑΛΑΙΟ 5

The import system

Python code in one *module* gains access to the code in another module by the process of *importing* it. The `import` statement is the most common way of invoking the import machinery, but it is not the only way. Functions such as `importlib.import_module()` and built-in `__import__()` can also be used to invoke the import machinery.

The `import` statement combines two operations; it searches for the named module, then it binds the results of that search to a name in the local scope. The search operation of the `import` statement is defined as a call to the `__import__()` function, with the appropriate arguments. The return value of `__import__()` is used to perform the name binding operation of the `import` statement. See the `import` statement for the exact details of that name binding operation.

A direct call to `__import__()` performs only the module search and, if found, the module creation operation. While certain side-effects may occur, such as the importing of parent packages, and the updating of various caches (including `sys.modules`), only the `import` statement performs a name binding operation.

When an `import` statement is executed, the standard builtin `__import__()` function is called. Other mechanisms for invoking the import system (such as `importlib.import_module()`) may choose to bypass `__import__()` and use their own solutions to implement import semantics.

When a module is first imported, Python searches for the module and if found, it creates a module object[1], initializing it. If the named module cannot be found, a `ModuleNotFoundError` is raised. Python implements various strategies to search for the named module when the import machinery is invoked. These strategies can be modified and extended by using various hooks described in the sections below.

Άλλαξε στην έκδοση 3.3: The import system has been updated to fully implement the second phase of **PEP 302**. There is no longer any implicit import machinery - the full import system is exposed through `sys.meta_path`. In addition, native namespace package support has been implemented (see **PEP 420**).

[1] See `types.ModuleType`.

5.1 importlib

The importlib module provides a rich API for interacting with the import system. For example importlib.import_module() provides a recommended, simpler API than built-in __import__() for invoking the import machinery. Refer to the importlib library documentation for additional detail.

5.2 Packages

Python has only one type of module object, and all modules are of this type, regardless of whether the module is implemented in Python, C, or something else. To help organize modules and provide a naming hierarchy, Python has a concept of *packages*.

You can think of packages as the directories on a file system and modules as files within directories, but don't take this analogy too literally since packages and modules need not originate from the file system. For the purposes of this documentation, we'll use this convenient analogy of directories and files. Like file system directories, packages are organized hierarchically, and packages may themselves contain subpackages, as well as regular modules.

It's important to keep in mind that all packages are modules, but not all modules are packages. Or put another way, packages are just a special kind of module. Specifically, any module that contains a __path__ attribute is considered a package.

All modules have a name. Subpackage names are separated from their parent package name by a dot, akin to Python's standard attribute access syntax. Thus you might have a package called email, which in turn has a subpackage called email.mime and a module within that subpackage called email.mime.text.

5.2.1 Regular packages

Python defines two types of packages, *regular packages* and *namespace packages*. Regular packages are traditional packages as they existed in Python 3.2 and earlier. A regular package is typically implemented as a directory containing an __init__.py file. When a regular package is imported, this __init__.py file is implicitly executed, and the objects it defines are bound to names in the package's namespace. The __init__.py file can contain the same Python code that any other module can contain, and Python will add some additional attributes to the module when it is imported.

For example, the following file system layout defines a top level parent package with three subpackages:

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

Importing parent.one will implicitly execute parent/__init__.py and parent/one/__init__.py. Subsequent imports of parent.two or parent.three will execute parent/two/__init__.py and parent/three/__init__.py respectively.

5.2.2 Namespace packages

A namespace package is a composite of various *portions*, where each portion contributes a subpackage to the parent package. Portions may reside in different locations on the file system. Portions may also be found in zip files, on the network, or anywhere else that Python searches during import. Namespace packages may or may not correspond directly to objects on the file system; they may be virtual modules that have no concrete representation.

Namespace packages do not use an ordinary list for their `__path__` attribute. They instead use a custom iterable type which will automatically perform a new search for package portions on the next import attempt within that package if the path of their parent package (or `sys.path` for a top level package) changes.

With namespace packages, there is no `parent/__init__.py` file. In fact, there may be multiple `parent` directories found during import search, where each one is provided by a different portion. Thus `parent/one` may not be physically located next to `parent/two`. In this case, Python will create a namespace package for the top-level `parent` package whenever it or one of its subpackages is imported.

See also **PEP 420** for the namespace package specification.

5.3 Searching

To begin the search, Python needs the *fully qualified* name of the module (or package, but for the purposes of this discussion, the difference is immaterial) being imported. This name may come from various arguments to the *import* statement, or from the parameters to the `importlib.import_module()` or `__import__()` functions.

This name will be used in various phases of the import search, and it may be the dotted path to a submodule, e.g. `foo.bar.baz`. In this case, Python first tries to import `foo`, then `foo.bar`, and finally `foo.bar.baz`. If any of the intermediate imports fail, a `ModuleNotFoundError` is raised.

5.3.1 The module cache

The first place checked during import search is `sys.modules`. This mapping serves as a cache of all modules that have been previously imported, including the intermediate paths. So if `foo.bar.baz` was previously imported, `sys.modules` will contain entries for `foo`, `foo.bar`, and `foo.bar.baz`. Each key will have as its value the corresponding module object.

During import, the module name is looked up in `sys.modules` and if present, the associated value is the module satisfying the import, and the process completes. However, if the value is `None`, then a `ModuleNotFoundError` is raised. If the module name is missing, Python will continue searching for the module.

`sys.modules` is writable. Deleting a key may not destroy the associated module (as other modules may hold references to it), but it will invalidate the cache entry for the named module, causing Python to search anew for the named module upon its next import. The key can also be assigned to `None`, forcing the next import of the module to result in a `ModuleNotFoundError`.

Beware though, as if you keep a reference to the module object, invalidate its cache entry in `sys.modules`, and then re-import the named module, the two module objects will *not* be the same. By contrast, `importlib.reload()` will reuse the *same* module object, and simply reinitialise the module contents by rerunning the module's code.

5.3.2 Finders and loaders

If the named module is not found in `sys.modules`, then Python's import protocol is invoked to find and load the module. This protocol consists of two conceptual objects, *finders* and *loaders*. A finder's job is to determine whether it can find the named module using whatever strategy it knows about. Objects that implement both of these interfaces are referred to as *importers* - they return themselves when they find that they can load the requested module.

Python includes a number of default finders and importers. The first one knows how to locate built-in modules, and the second knows how to locate frozen modules. A third default finder searches an *import path* for modules. The *import path* is a list of locations that may name file system paths or zip files. It can also be extended to search for any locatable resource, such as those identified by URLs.

The import machinery is extensible, so new finders can be added to extend the range and scope of module searching.

Finders do not actually load modules. If they can find the named module, they return a *module spec*, an encapsulation of the module's import-related information, which the import machinery then uses when loading the module.

The following sections describe the protocol for finders and loaders in more detail, including how you can create and register new ones to extend the import machinery.

Άλλαξε στην έκδοση 3.4: In previous versions of Python, finders returned *loaders* directly, whereas now they return module specs which *contain* loaders. Loaders are still used during import but have fewer responsibilities.

5.3.3 Import hooks

The import machinery is designed to be extensible; the primary mechanism for this are the *import hooks*. There are two types of import hooks: *meta hooks* and *import path hooks*.

Meta hooks are called at the start of import processing, before any other import processing has occurred, other than `sys.modules` cache look up. This allows meta hooks to override `sys.path` processing, frozen modules, or even built-in modules. Meta hooks are registered by adding new finder objects to `sys.meta_path`, as described below.

Import path hooks are called as part of `sys.path` (or `package.__path__`) processing, at the point where their associated path item is encountered. Import path hooks are registered by adding new callables to `sys.path_hooks` as described below.

5.3.4 The meta path

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

If the meta path finder knows how to handle the named module, it returns a spec object. If it cannot handle the named module, it returns `None`. If `sys.meta_path` processing reaches the end of its list without returning a spec, then a `ModuleNotFoundError` is raised. Any other exceptions raised are simply propagated up, aborting the import process.

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

The meta path may be traversed multiple times for a single import request. For example, assuming none of the modules involved has already been cached, importing `foo.bar.baz` will first perform a top level import, calling `mpf.find_spec("foo", None, None)` on each meta path finder (`mpf`). After `foo` has been imported, `foo.bar` will be imported by traversing the meta path a second time, calling `mpf.find_spec("foo.bar", foo.__path__, None)`. Once `foo.bar` has been imported, the final traversal will call `mpf.find_spec("foo.bar.baz", foo.bar.__path__, None)`.

Some meta path finders only support top level imports. These importers will always return `None` when anything other than `None` is passed as the second argument.

Python's default `sys.meta_path` has three meta path finders, one that knows how to import built-in modules, one that knows how to import frozen modules, and one that knows how to import modules from an *import path* (i.e. the *path based finder*).

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

5.4 Loading

If and when a module spec is found, the import machinery will use it (and the loader it contains) when loading the module. Here is an approximation of what happens during the loading portion of import:

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

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

Note the following details:

- If there is an existing module object with the given name in `sys.modules`, import will have already returned it.

- The module will exist in `sys.modules` before the loader executes the module code. This is crucial because the module code may (directly or indirectly) import itself; adding it to `sys.modules` beforehand prevents unbounded recursion in the worst case and multiple loading in the best.
- If loading fails, the failing module – and only the failing module – gets removed from `sys.modules`. Any module already in the `sys.modules` cache, and any module that was successfully loaded as a side-effect, must remain in the cache. This contrasts with reloading where even the failing module is left in `sys.modules`.
- After the module is created but before execution, the import machinery sets the import-related module attributes («_init_module_attrs» in the pseudo-code example above), as summarized in a *later section*.
- Module execution is the key moment of loading in which the module's namespace gets populated. Execution is entirely delegated to the loader, which gets to decide what gets populated and how.
- The module created during loading and passed to exec_module() may not be the one returned at the end of import[2].

Άλλαξε στην έκδοση 3.4: The import system has taken over the boilerplate responsibilities of loaders. These were previously performed by the `importlib.abc.Loader.load_module()` method.

5.4.1 Loaders

Module loaders provide the critical function of loading: module execution. The import machinery calls the `importlib.abc.Loader.exec_module()` method with a single argument, the module object to execute. Any value returned from `exec_module()` is ignored.

Loaders must satisfy the following requirements:

- If the module is a Python module (as opposed to a built-in module or a dynamically loaded extension), the loader should execute the module's code in the module's global name space (`module.__dict__`).
- If the loader cannot execute the module, it should raise an `ImportError`, although any other exception raised during `exec_module()` will be propagated.

In many cases, the finder and loader can be the same object; in such cases the `find_spec()` method would just return a spec with the loader set to `self`.

Module loaders may opt in to creating the module object during loading by implementing a `create_module()` method. It takes one argument, the module spec, and returns the new module object to use during loading. `create_module()` does not need to set any attributes on the module object. If the method returns `None`, the import machinery will create the new module itself.

Νέο στην έκδοση 3.4: The `create_module()` method of loaders.

Άλλαξε στην έκδοση 3.4: The `load_module()` method was replaced by `exec_module()` and the import machinery assumed all the boilerplate responsibilities of loading.

For compatibility with existing loaders, the import machinery will use the `load_module()` method of loaders if it exists and the loader does not also implement `exec_module()`. However, `load_module()` has been deprecated and loaders should implement `exec_module()` instead.

The `load_module()` method must implement all the boilerplate loading functionality described above in addition to executing the module. All the same constraints apply, with some additional clarification:

- If there is an existing module object with the given name in `sys.modules`, the loader must use that existing module. (Otherwise, `importlib.reload()` will not work correctly.) If the named module does not exist in `sys.modules`, the loader must create a new module object and add it to `sys.modules`.

[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.

- The module *must* exist in `sys.modules` before the loader executes the module code, to prevent unbounded recursion or multiple loading.
- If loading fails, the loader must remove any modules it has inserted into `sys.modules`, but it must remove **only** the failing module(s), and only if the loader itself has loaded the module(s) explicitly.

Άλλαξε στην έκδοση 3.5: A `DeprecationWarning` is raised when `exec_module()` is defined but `create_module()` is not.

Άλλαξε στην έκδοση 3.6: An `ImportError` is raised when `exec_module()` is defined but `create_module()` is not.

5.4.2 Submodules

When a submodule is loaded using any mechanism (e.g. `importlib` APIs, the `import` or `import-from` statements, or built-in `__import__()`) a binding is placed in the parent module's namespace to the submodule object. For example, if package `spam` has a submodule `foo`, after importing `spam.foo`, `spam` will have an attribute `foo` which is bound to the submodule. Let's say you have the following directory structure:

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

and `spam/__init__.py` has the following line in it:

```
from .foo import Foo
```

then executing the following puts name bindings for `foo` and `Foo` in the `spam` module:

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

Given Python's familiar name binding rules this might seem surprising, but it's actually a fundamental feature of the import system. The invariant holding is that if you have `sys.modules['spam']` and `sys.modules['spam.foo']` (as you would after the above import), the latter must appear as the `foo` attribute of the former.

5.4.3 Module spec

The import machinery uses a variety of information about each module during import, especially before loading. Most of the information is common to all modules. The purpose of a module's spec is to encapsulate this import-related information on a per-module basis.

Using a spec during import allows state to be transferred between import system components, e.g. between the finder that creates the module spec and the loader that executes it. Most importantly, it allows the import machinery to perform the boilerplate operations of loading, whereas without a module spec the loader had that responsibility.

The module's spec is exposed as the `__spec__` attribute on a module object. See `ModuleSpec` for details on the contents of the module spec.

Νέο στην έκδοση 3.4.

5.4.4 Import-related module attributes

The import machinery fills in these attributes on each module object during loading, based on the module's spec, before the loader executes the module.

`__name__`
The `__name__` attribute must be set to the fully-qualified name of the module. This name is used to uniquely identify the module in the import system.

`__loader__`
The `__loader__` attribute must be set to the loader object that the import machinery used when loading the module. This is mostly for introspection, but can be used for additional loader-specific functionality, for example getting data associated with a loader.

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

This attribute is used instead of `__name__` to calculate explicit relative imports for main modules, as defined in **PEP 366**. It is expected to have the same value as `__spec__.parent`.

Άλλαξε στην έκδοση 3.6: The value of `__package__` is expected to be the same as `__spec__.parent`.

`__spec__`
The `__spec__` attribute must be set to the module spec that was used when importing the module. Setting `__spec__` appropriately applies equally to *modules initialized during interpreter startup*. The one exception is `__main__`, where `__spec__` is *set to None in some cases*.

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

Νέο στην έκδοση 3.4.

Άλλαξε στην έκδοση 3.6: `__spec__.parent` is used as a fallback when `__package__` is not defined.

`__path__`
If the module is a package (either regular or namespace), the module object's `__path__` attribute must be set. The value must be iterable, but may be empty if `__path__` has no further significance. If `__path__` is not empty, it must produce strings when iterated over. More details on the semantics of `__path__` are given *below*.

Non-package modules should not have a `__path__` attribute.

`__file__`

`__cached__`
`__file__` is optional. If set, this attribute's value must be a string. The import system may opt to leave `__file__` unset if it has no semantic meaning (e.g. a module loaded from a database).

If `__file__` is set, it may also be appropriate to set the `__cached__` attribute which is the path to any compiled version of the code (e.g. 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**).

It is also appropriate to set `__cached__` when `__file__` is not set. However, that scenario is quite atypical. Ultimately, the loader is what makes use of `__file__` and/or `__cached__`. So if a loader can load from a cached module but otherwise does not load from a file, that atypical scenario may be appropriate.

5.4.5 module.__path__

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`.

`__path__` must be an iterable of strings, but it may be empty. The same rules used for `sys.path` also apply to a package's `__path__`, and `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.

5.4.6 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.

Άλλαξε στην έκδοση 3.4: Use of `loader.module_repr()` has been deprecated and the module spec is now used by the import machinery to generate a module repr.

For backward compatibility with Python 3.3, the module repr will be generated by calling the loader's `module_repr()` method, if defined, before trying either approach described above. However, the method is deprecated.

5.4.7 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.

Άλλαξε στην έκδοση 3.7: 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 and bytes should be present on `sys.path`; all other data types are ignored. The encoding of bytes entries is determined by the individual *path entry finders*.

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[3].

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.

Άλλαξε στην έκδοση 3.4: find_spec() replaced find_loader() and find_module(), both of which are now deprecated, but will be used if find_spec() is not defined.

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).

[3] In legacy code, it is possible to find instances of imp.NullImporter in the sys.path_importer_cache. It is recommended that code be changed to use None instead. See portingpythoncode for more details.

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()`.

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` option
- 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 Open issues

XXX It would be really nice to have a diagram.

XXX * (import_machinery.rst) how about a section devoted just to the attributes of modules and packages, perhaps expanding upon or supplanting the related entries in the data model reference page?

XXX runpy, pkgutil, et al in the library manual should all get «See Also» links at the top pointing to the new import system section.

XXX Add more explanation regarding the different ways in which `__main__` is initialized?

XXX Add more info on `__main__` quirks/pitfalls (i.e. copy from **PEP 395**).

5.10 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.

ΚΕΦΑΛΑΙΟ 6

Expressions

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 *Ονομασία και σύνδεση* 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. Private names are transformed to a longer form before code is generated for them. The transformation inserts the class name, with leading underscores removed and a single underscore inserted, in front of the name. For example, the identifier `__spam` occurring in a class named `Ham` will be transformed to `_Ham__spam`. This transformation is independent of the syntactical context in which the identifier is used. If the transformed name is extremely long (longer than 255 characters), implementation defined truncation may happen. If the class name consists only of underscores, no transformation is done.

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 operator. 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 either `async for` clauses or `await` expressions 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**.

Νέο στην έκδοση 3.6: Asynchronous comprehensions were introduced.

Άλλαξε στην έκδοση 3.8: `yield` and `yield from` prohibited in the implicitly nested scope.

6.2.5 List displays

A list display is a possibly empty series of expressions enclosed in square brackets:

```
list_display ::= "[" [starred_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 ::= "{" (starred_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 key/datum pairs enclosed in curly braces:

```
dict_display       ::=  "{" [key_datum_list | dict_comprehension] "}"
key_datum_list     ::=  key_datum ("," key_datum)* [","]
key_datum          ::=  expression ":" expression | "**" or_expr
dict_comprehension ::=  expression ":" expression comp_for
```

A dictionary display yields a new dictionary object.

If a comma-separated sequence of key/datum pairs 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 datum. This means that you can specify the same key multiple times in the key/datum 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 key/datum pairs and earlier dictionary unpackings.

Νέο στην έκδοση 3.5: 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 *The standard type hierarchy*. (To summarize, the key type should be *hashable*, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last datum (textually rightmost in the display) stored for a given key value prevails.

Άλλαξε στην έκδοση 3.8: 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*).

Νέο στην έκδοση 3.6: Asynchronous generator expressions were introduced.

Άλλαξε στην έκδοση 3.7: 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.

Άλλαξε στην έκδοση 3.8: `yield` and `yield from` prohibited in the implicitly nested scope.

6.2.9 Yield expressions

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

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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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.

Άλλαξε στην έκδοση 3.8: 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 *Asynchronous generator functions*.

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 `expression_list` to the generator's caller. 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).

Άλλαξε στην έκδοση 3.3: 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.

Δείτε επίσης:

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 `expression_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 compatability, 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.

`generator.close()`

Raises a `GeneratorExit` at the point where the generator function was paused. If the generator function then exits gracefully, is already closed, or raises `GeneratorExit` (by not catching the exception), close returns to its caller. If the generator yields a value, a `RuntimeError` is raised. If the generator raises any other exception, it is propagated to the caller. `close()` does nothing if the generator has already exited due to an exception or normal exit.

Examples

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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
...     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.»

Asynchronous generator functions

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 *expression_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.

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, 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.

coroutine agen.**__anext__**()

Returns an awaitable which when run starts to execute the asynchronous generator or resumes it at the last executed yield expression. When an asynchronous generator function is resumed with an `__anext__()` method, the current yield expression always evaluates to `None` in the returned awaitable, which when run will continue to the next yield expression. The value of the `expression_list` of the yield expression is the value of the `StopIteration` exception raised by the completing coroutine. If the asynchronous generator exits without yielding another value, the awaitable instead raises a `StopAsyncIteration` exception, signalling that the asynchronous iteration has completed.

This method is normally called implicitly by a `async for` loop.

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

Returns an awaitable which when run resumes the execution of the asynchronous generator. As with the `send()` method for a generator, this «sends» a value into the asynchronous generator function, and the *value* argument becomes the result of the current yield expression. The awaitable returned by the `asend()` method will return the next value yielded by the generator as the value of the raised `StopIteration`, or raises `StopAsyncIteration` if the asynchronous generator exits without yielding another value. When `asend()` is called to start the asynchronous generator, it must be called with `None` as the argument, because there is no yield expression that could receive the value.

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

Returns an awaitable that raises an exception of type `type` at the point where the asynchronous generator was paused, and returns the next value yielded by the generator function as the value of the raised `StopIteration` exception. If the asynchronous generator exits without yielding another value, a `StopAsyncIteration` exception is raised by the awaitable. If the generator function does not catch the passed-in exception, or raises a different exception, then when the awaitable is run that exception propagates to the caller of the awaitable.

coroutine agen.**aclose**()

Returns an awaitable that when run will throw a `GeneratorExit` into the asynchronous generator function at the point where it was paused. If the asynchronous generator function then exits gracefully, is already closed, or raises `GeneratorExit` (by not catching the exception), then the returned awaitable will raise a `StopIteration` exception. Any further awaitables returned by subsequent calls to the asynchronous generator will raise a `StopAsyncIteration` exception. If the asynchronous generator yields a value, a `RuntimeError` is raised by the awaitable. If the asynchronous generator raises any other exception, it is propagated to the caller of the awaitable. If the asynchronous generator has already exited due to an exception or normal exit, then further calls to `aclose()` will return an awaitable that does nothing.

6.3 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. This production can be customized by overriding the `__getattr__()` method. If this attribute is not available, the exception `AttributeError` is raised. Otherwise, the type and value of the object produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

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 "[" expression_list "]"
```

When an object is subscripted, the interpreter will evaluate the primary and the expression list.

The primary must evaluate to an object that supports subscription. An object may support subscription through defining one or both of `__getitem__()` and `__class_getitem__()`. When the primary is subscripted, the evaluated result of the expression list will be passed to one of these methods. For more details on when `__class_getitem__` is called instead of `__getitem__`, see *__class_getitem__ versus __getitem__*.

If the expression list contains at least one comma, it will evaluate to a `tuple` containing the items of the expression list. Otherwise, the expression list will evaluate to the value of the list's sole member.

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 *The standard type hierarchy*) 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 *Function definitions* 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 value of 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 implementation detail: 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 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 keyword is already present (as an explicit keyword argument, or from another unpacking), 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.

Άλλαξε στην έκδοση 3.5: 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 *Function definitions*. When the code block executes a `return` statement, this specifies the return value of the function call.

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
```

Νέο στην έκδοση 3.5.

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__()` method.

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.

Νέο στην έκδοση 3.5.

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.

This operation can be customized using the special `__truediv__()` and `__floordiv__()` 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].

[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

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__()` method.

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__()` method.

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.

This operation can be customized using the special `__lshift__()` and `__rshift__()` 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.

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`.

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 *Objects, values and types* 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).

  – 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.

[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()`.

- 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` and `y == x`

  `x != y` and `y != x`

  `x < y` and `y > x`

  `x <= y` and `y >= x`

- Comparison should be transitive. The following (non-exhaustive) examples illustrate that:

  `x > y and y > z` implies `x > z`

  `x < y and y <= z` implies `x < z`

- Inverse comparison should result in the boolean negation. In other words, the following expressions should have the same result:

  `x == y` and `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.

Νέο στην έκδοση 3.8: 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 *Function definitions* for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements or annotations.

6.15 Expression lists

```
expression_list    ::= expression ("," expression)* [","]
starred_list       ::= starred_item ("," starred_item)* [","]
starred_expression ::= expression | (starred_item ",")* [starred_item]
starred_item       ::= assignment_expression | "*" or_expr
```

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.

Νέο στην έκδοση 3.5: Iterable unpacking in expression lists, originally proposed by **PEP 448**.

The trailing comma is required only to create a single tuple (a.k.a. a *singleton*); 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, which groups 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.

Operator	Description
`(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

[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.

ΚΕΦΑΛΑΙΟ 7

Simple statements

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
```

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 *The standard type hierarchy*).

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/datum 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 implementation detail: 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)
```

Δείτε επίσης:

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 expression 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 single target is allowed.

For simple names as 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.

For expressions as assignment targets, the annotations are 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.

Δείτε επίσης:

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.

Άλλαξε στην έκδοση 3.8: Now annotated assignments allow same expressions in the right hand side as the regular assignments. Previously, some expressions (like un-parenthesized tuple expressions) caused a syntax error.

7.3 The `assert` statement

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 The `pass` statement

```
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 The `del` statement

```
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).

Άλλαξε στην έκδοση 3.2: 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 The `return` statement

```
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 The `yield` statement

```
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 The `raise` statement

```
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, which is writable. 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>
ZeroDivisionError: division by zero

The above exception was the direct cause of the following exception:

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

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>
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

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

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 *Εξαιρέσεις*, and information about handling exceptions is in section *The try statement*.

Άλλαξε στην έκδοση 3.3: `None` is now permitted as `Y` in `raise X from Y`.

Νέο στην έκδοση 3.3: The `__suppress_context__` attribute to suppress automatic display of the exception context.

7.9 The `break` statement

```
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 The `continue` statement

```
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 The `import` statement

```
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

Examples:

```
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.

Raises an auditing event `import` with arguments `module`, `filename`, `sys.path`, `sys.meta_path`, `sys.path_hooks`.

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.

Δείτε επίσης:

PEP 236 - Back to the __future__ The original proposal for the __future__ mechanism.

7.12 The `global` statement

```
global_stmt  ::=  "global" identifier ("," identifier)*
```

The *`global`* statement is a declaration which holds for the entire current code block. It means that the listed identifiers are 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.

Names listed in a *`global`* statement must not be used in the same code block textually preceding that `global` statement.

Names listed in a *global* statement must not be defined as formal parameters or in a *for* loop control target, *class* definition, function definition, *import* statement, or variable annotation.

CPython implementation detail: The current implementation does not enforce some of these restrictions, but programs should not abuse this freedom, as future implementations may enforce them or silently change the meaning of the program.

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 The `nonlocal` statement

```
nonlocal_stmt ::=  "nonlocal" identifier ("," identifier)*
```

The *nonlocal* statement causes the listed identifiers to refer to previously bound variables in the nearest enclosing scope excluding globals. This is important because the default behavior for binding is to search the local namespace first. The statement allows encapsulated code to rebind variables outside of the local scope besides the global (module) scope.

Names listed in a *nonlocal* statement, unlike those listed in a *global* statement, must refer to pre-existing bindings in an enclosing scope (the scope in which a new binding should be created cannot be determined unambiguously).

Names listed in a *nonlocal* statement must not collide with pre-existing bindings in the local scope.

Δείτε επίσης:

PEP 3104 - Access to Names in Outer Scopes The specification for the *nonlocal* statement.

ΚΕΦΑΛΑΙΟ 8

Compound statements

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
                    | 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 The `if` statement

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 The `while` statement

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 The `for` statement

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" expression_list ":" suite
              ["else" ":" suite]
```

The expression list is evaluated once; it should yield an iterable object. An iterator is created for the result of the `expression_list`. The suite is then executed once for each item provided by the iterator, in the order returned by the iterator. Each item in turn is assigned to the target list using the standard rules for assignments (see *Assignment statements*), and then the suite is executed. When the items are exhausted (which is immediately when the sequence is empty or an iterator raises a `StopIteration` exception), 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 function `range()` returns an iterator of integers suitable to emulate the effect of Pascal's `for i := a to b do`; e.g., `list(range(3))` returns the list `[0, 1, 2]`.

Σημείωση: There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, e.g. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop. This can lead to nasty bugs that can be avoided by making a temporary copy using a slice of the whole sequence, e.g.,

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

8.4 The `try` statement

The *try* statement specifies exception handlers and/or cleanup code for a group of statements:

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

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, that expression is evaluated, and the clause matches the exception if the resulting object is «compatible» with the exception. An object is compatible with an exception if the object is the class or a *non-virtual base class* of the exception object, or a tuple containing an item that is the class or a non-virtual base class of the exception object.

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
    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, details about the exception are stored in the `sys` module and can be accessed via `sys.exc_info()`. `sys.exc_info()` returns a 3-tuple consisting of the exception class, the exception instance and a traceback object (see section *The standard type hierarchy*) identifying the point in the program where the exception occurred. `sys.exc_info()` values are restored to their previous values (before the call) when returning from a function that handled an exception.

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.

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
```

[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.

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'
```

Additional information on exceptions can be found in section *Εξαιρέσεις*, and information on using the *raise* statement to generate exceptions may be found in section *The raise statement*.

Άλλαξε στην έκδοση 3.8: Prior to Python 3.8, a *continue* statement was illegal in the *finally* clause due to a problem with the implementation.

8.5 The with statement

The *with* statement is used to wrap the execution of a block with methods defined by a context manager (see section *With Statement Context Managers*). This allows common *try*...*except*...*finally* usage patterns to be encapsulated for convenient reuse.

```
with_stmt  ::=  "with" with_item ("," with_item)* ":" suite
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.

   Σημείωση: 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 6 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.

The following code:

```
with EXPRESSION as TARGET:
    SUITE
```

is semantically equivalent to:

```
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
```

is semantically equivalent to:

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

Άλλαξε στην έκδοση 3.1: Support for multiple context expressions.

Δείτε επίσης:

PEP 343 - The «with» statement The specification, background, and examples for the Python *with* statement.

8.6 Function definitions

A function definition defines a user-defined function object (see section *The standard type hierarchy*):

```
funcdef                    ::=  [decorators] "def" funcname "(" [parameter_list] ")"
                                ["->" expression] ":" suite
decorators                 ::=  decorator+
decorator                  ::=  "@" assignment_expression NEWLINE
parameter_list             ::=  defparameter ("," defparameter)* "," "/" ["," [parameter_l
                                | parameter_list_no_posonly
parameter_list_no_posonly  ::=  defparameter ("," defparameter)* ["," [parameter_list_star
```

```
                                 | parameter_list_starargs
parameter_list_starargs     ::=  "*" [parameter] ("," defparameter)* ["," ["**" parameter
                                 | "**" parameter [","]
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.[2]

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
```

is roughly equivalent to

```
def func(): pass
func = f1(arg)(f2(func))
```

except that the original function is not temporarily bound to the name `func`.

Άλλαξε στην έκδοση 3.9: Functions may be decorated with any valid `assignment_expression`. Previously, the grammar was much more restrictive; see **PEP 614** for details.

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 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 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.:

```
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`»

[2] 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*.

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.

Άλλαξε στην έκδοση 3.8: 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`. 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.

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 *Ονομασία και σύνδεση* for details.

Δείτε επίσης:

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.

8.7 Class definitions

A class definition defines a class object (see section *The standard type hierarchy*):

```
classdef    ::=  [decorators] "class" classname [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 *Ονομασία και σύνδεση*), 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.[3] 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
```

is roughly equivalent to

```
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.

Άλλαξε στην έκδοση 3.9: Classes may be decorated with any valid *assignment_expression*. Previously, the grammar was much more restrictive; see **PEP 614** for details.

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.

Δείτε επίσης:

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 - Class Decorators The proposal that added class decorators. Function and method decorators were introduced in **PEP 318**.

8.8 Coroutines

Νέο στην έκδοση 3.5.

[3] 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*.

8.8.1 Coroutine function definition

```
async_funcdef ::=  [decorators] "async" "def" funcname "(" [parameter_list] ")"
                   ["->" expression] ":" suite
```

Execution of Python coroutines can be suspended and resumed at many points (see *coroutine*). Inside the body of a coroutine function, `await` and `async` identifiers become reserved keywords; `await` expressions, `async for` and `async with` can only be used in coroutine function bodies.

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.

An example of a coroutine function:

```
async def func(param1, param2):
    do_stuff()
    await some_coroutine()
```

8.8.2 The `async for` statement

```
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.

The following code:

```
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
```

See also `__aiter__()` and `__anext__()` for details.

It is a `SyntaxError` to use an `async for` statement outside the body of a coroutine function.

8.8.3 The `async with` statement

```
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.

The following code:

```
async with EXPRESSION as TARGET:
    SUITE
```

is semantically equivalent to:

```
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)
```

See also `__aenter__()` and `__aexit__()` for details.

It is a `SyntaxError` to use an `async with` statement outside the body of a coroutine function.

Δείτε επίσης:

PEP 492 - Coroutines with async and await syntax The proposal that made coroutines a proper standalone concept in Python, and added supporting syntax.

ΚΕΦΑΛΑΙΟ 9

Top-level components

The Python interpreter can get its input from a number of sources: from a script passed to it as standard input or as program argument, typed in interactively, from a module source file, etc. This chapter gives the syntax used in these cases.

9.1 Complete Python programs

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for `sys` (various system services), `builtins` (built-in functions, exceptions and `None`) and `__main__`. The latter is used to provide the local and global namespace for execution of the complete program.

The syntax for a complete Python program is that for file input, described in the next section.

The interpreter may also be invoked in interactive mode; in this case, it does not read and execute a complete program but reads and executes one statement (possibly compound) at a time. The initial environment is identical to that of a complete program; each statement is executed in the namespace of `__main__`.

A complete program can be passed to the interpreter in three forms: with the `-c` *string* command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.

9.2 File input

All input read from non-interactive files has the same form:

```
file_input ::=  (NEWLINE | statement)*
```

This syntax is used in the following situations:

- when parsing a complete Python program (from a file or from a string);
- when parsing a module;
- when parsing a string passed to the `exec()` function;

9.3 Interactive input

Input in interactive mode is parsed using the following grammar:

```
interactive_input ::=  [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Note that a (top-level) compound statement must be followed by a blank line in interactive mode; this is needed to help the parser detect the end of the input.

9.4 Expression input

`eval()` is used for expression input. It ignores leading whitespace. The string argument to `eval()` must have the following form:

```
eval_input ::=  expression_list NEWLINE*
```

ΚΕΦΑΛΑΙΟ 10

Πλήρης προδιαγραφή γραμματικής

Αυτή είναι η πλήρης γραμματική της Python, που προέρχεται απευθείας από τη γραμματική που χρησιμοποιείται για τη δημιουργία του αναλυτή CPython (βλ. Grammar/python.gram). Η έκδοση αυτή παραλείπει λεπτομέρειες που σχετίζονται με τη δημιουργία κώδικα και την ανάκτηση από σφάλματα.

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).

```
# PEG grammar for Python


file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER
fstring: star_expressions

# type_expressions allow */** but ignore them
type_expressions:
    | ','.expression+ ',' '*' expression ',' '**' expression
    | ','.expression+ ',' '*' expression
    | ','.expression+ ',' '**' expression
    | '*' expression ',' '**' expression
    | '*' expression
    | '**' expression
    | ','.expression+

statements: statement+
statement: compound_stmt  | simple_stmt
statement_newline:
    | compound_stmt NEWLINE
    | simple_stmt
    | NEWLINE
    | ENDMARKER
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
simple_stmt:
    | small_stmt !';' NEWLINE  # Not needed, there for speedup
    | ';'.small_stmt+ [';'] NEWLINE
# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
small_stmt:
    | assignment
    | 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

# 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)

augassign:
    | '+='
    | '-='
    | '*='
    | '@='
    | '/='
    | '%='
    | '&='
    | '|='
    | '^='
    | '<<='
    | '>>='
    | '**='
    | '//='

global_stmt: 'global' ','.NAME+
nonlocal_stmt: 'nonlocal' ','.NAME+

yield_stmt: yield_expr
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
assert_stmt: 'assert' expression [',' expression ]

del_stmt:
    | 'del' del_targets &(';' | NEWLINE)

import_stmt: import_name | import_from
import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from:
    | 'from' ('.' | '...')* dotted_name 'import' import_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

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_stmt:
    | 'while' named_expression ':' block [else_block]

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_stmt:
    | 'with' '(' ','.with_item+ ','? ')' ':' block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | ASYNC 'with' '(' ','.with_item+ ','? ')' ':' block
    | ASYNC 'with' ','.with_item+ ':' [TYPE_COMMENT] block
with_item:
    | expression 'as' star_target &(',' | ')' | ':')
    | expression

try_stmt:
    | 'try' ':' block finally_block
    | 'try' ':' block except_block+ [else_block] [finally_block]
except_block:
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    | 'except' expression ['as' NAME ] ':' block
    | 'except' ':' block
finally_block: 'finally' ':' block

return_stmt:
    | 'return' [star_expressions]

raise_stmt:
    | 'raise' expression ['from' expression ]
    | 'raise'

function_def:
    | decorators function_def_raw
    | function_def_raw

function_def_raw:
    | 'def' NAME '(' [params] ')' ['->' expression ] ':' [func_type_comment] block
    | ASYNC 'def' NAME '(' [params] ')' ['->' expression ] ':' [func_type_comment]␣
↪block
func_type_comment:
    | NEWLINE TYPE_COMMENT &(NEWLINE INDENT)   # Must be followed by indented block
    | TYPE_COMMENT

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_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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
#
# 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_with_default:
    | param default ',' TYPE_COMMENT?
    | param default TYPE_COMMENT? &')'
param_maybe_default:
    | param default? ',' TYPE_COMMENT?
    | param default? TYPE_COMMENT? &')'
param: NAME annotation?

annotation: ':' expression
default: '=' expression

decorators: ('@' named_expression NEWLINE )+

class_def:
    | decorators class_def_raw
    | class_def_raw
class_def_raw:
    | 'class' NAME ['(' [arguments] ')' ] ':' block

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmt

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
named_expression:
    | NAME ':=' ~ expression
    | expression !':='

annotated_rhs: yield_expr | star_expressions

expressions:
    | expression (',' expression )+ [',']
    | expression ','
    | expression
expression:
    | disjunction 'if' disjunction 'else' expression
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    | disjunction
    | lambdef

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

disjunction:
    | conjunction ('or' conjunction )+
    | conjunction
conjunction:
    | inversion ('and' inversion )+
    | inversion
inversion:
    | 'not' inversion
    | comparison
comparison:
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    | 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_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

sum:
    | sum '+' term
    | sum '-' term
    | term
term:
    | term '*' factor
    | term '/' factor
    | term '//' factor
    | term '%' factor
    | term '@' factor
    | factor
factor:
    | '+' factor
    | '-' factor
    | '~' factor
    | power
power:
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
    | await_primary '**' factor
    | await_primary
await_primary:
    | AWAIT primary
    | primary
primary:
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice !','
    | ','.slice+ [',']
slice:
    | [expression] ':' [expression] [':' [expression] ]
    | expression
atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | '__peg_parser__'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

strings: STRING+
list:
    | '[' [star_named_expressions] ']'
listcomp:
    | '[' named_expression ~ for_if_clauses ']'
tuple:
    | '(' [star_named_expression ',' [star_named_expressions]  ] ')'
group:
    | '(' (yield_expr | named_expression) ')'
genexp:
    | '(' named_expression ~ for_if_clauses ')'
set: '{' star_named_expressions '}'
setcomp:
    | '{' named_expression ~ for_if_clauses '}'
dict:
    | '{' [double_starred_kvpairs] '}'
dictcomp:
    | '{' kvpair for_if_clauses '}'
double_starred_kvpairs: ','.double_starred_kvpair+ [',']
double_starred_kvpair:
    | '**' bitwise_or
    | kvpair
kvpair: expression ':' expression
for_if_clauses:
    | for_if_clause+
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
for_if_clause:
    | ASYNC 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
    | 'for' star_targets 'in' ~ disjunction ('if' disjunction )*

yield_expr:
    | 'yield' 'from' expression
    | 'yield' [star_expressions]

arguments:
    | args [','] &')'
args:
    | ','.(starred_expression | named_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

# 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

del_targets: ','.del_target+ [',']
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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] ']'

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: '(' | '[' | '.'
```

ΠΑΡΑΡΤΗΜΑ Α′

Γλωσσάρι

>>> Το προεπιλεγμένο Python prompt του διαδραστικού shell. Συχνά εμφανίζεται για παραδείγματα κώδικα που μπορούν να εκτελεστούν διαδραστικά στον interpreter.

... Μπορεί να αναφέρεται σε:

- Το προεπιλεγμένο Python prompt του διαδραστικού shell κατά την εισαγωγή του κώδικα για ένα μπλοκ κώδικα με εσοχή, όταν βρίσκεται μέσα σε ένα ζεύγος ταιριασμένων αριστερών και δεξιών delimiters (παρενθέσεις, αγκύλες, άγκιστρα ή τριπλά εισαγωγικά), ή μετά τον καθορισμό ενός decorator.
- Η ενσωματωμένη σταθερά `Ellipsis`.

2to3 Ένα εργαλείο που προσπαθεί να μετατρέψει τον κώδικα Python 2.x σε κώδικα Python 3.x διαχειρίζοντας τις περισσότερες ασυμβατότητες που μπορούν να εντοπιστούν αναλύοντας την πηγή και διασχίζοντας το δέντρο ανάλυσης.

2to3 είναι διαθέσιμο στην στάνταρ βιβλιοθήκη ως `lib2to3`, παρέχεται ένα σημείο εισόδου ως `Tools/scripts/2to3`. Βλ. 2to3-reference.

αφηρημένη βασική κλάση Οι αφηρημένες βασικές κλάσεις συμπληρώνουν το *duck-typing* παρέχοντας έναν τρόπο ορισμού interfaces όταν άλλες τεχνικές όπως η `hasattr()` θα ήταν αδέξιες ή ανεπαίσθητα λανθασμένες (για παράδειγμα με *magic methods*). Τα ABC (abstract base class) εισάγουν εικονικές υποκλάσεις, οι οποίες είναι κλάσεις που δεν κληρονομούνται από μια κλάση, αλλά εξακολουθούν να αναγνωρίζονται από το `isinstance()` και από το `issubclass()`" βλ. την τεκμηρίωση του module `abc`. Η Python διαθέτει πολλά ενσωματωμένα ABC για δομές δεδομένων (στο module `collections.abc`), αριθμούς (στο module `numbers`), ροές (στο module μονάδα `io`), εισαγωγή finders και loaders (στο module `importlib.abc`). Μπορείτε να δημιουργήσετε τα δικά σας ABC με το module `abc`.

annotation Μια ετικέτα που σχετίζεται με μια μεταβλητή, ένα χαρακτηριστικό κλάσης ή μια παράμετρος συνάρτησης ή τιμή που επιστρέφεται, που χρησιμοποιείται κατά σύμβαση ως *type hint*.

Δεν είναι δυνατή η πρόσβαση στα annotations των τοπικών μεταβλητών κατά το χρόνο εκτέλεσης, αλλά τα annotations των global μεταβλητών, των χαρακτηριστικών κλάσης και των συναρτήσεων αποθηκεύονται στο ειδικό χαρακτηριστικό `__annotations__` των modules, των κλάσεων και των συναρτήσεων, αντίστοιχα.

See *variable annotation*, *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality.

όρισμα Μια τιμή μεταβιβάζεται σε μία *function* (ή *method*) κατά την κλήση της συνάρτησης. Υπάρχουν δύο είδη ορισμάτων:

- *keyword argument*: ένα όρισμα πριν από ένα αναγνωριστικό (π.χ. `name=`) σε μια κλήση συνάρτησης ή περνώντας το ως τιμή σε ένα λεξικό πριν από `**`. Για παράδειγμα, το `3` και το `5` αποτελούν ορίσματα λέξεων-κλειδιών στις ακόλουθες κλήσεις προς `complex()`:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *positional argument*: ένα όρισμα που δεν είναι όρισμα keyword. Τα ορίσματα θέσης μπορούν να εμφανίζονται στην αρχής μιας λίστας ορισμάτων ή/και να μεταβιβάζονται ως στοιχεία ενός *iterable* πριν από `*`. Για παράδειγμα, το `3` και το `5` αποτελούν ορίσματα θέσης στις παρακάτω κλήσεις:

```
complex(3, 5)
complex(*(3, 5))
```

Τα ορίσματα εκχωρούνται στις ονομασμένες τοπικές μεταβλητές στο σώμα μια συνάρτησης. Βλ. την ενότητα *Calls* για τους κανόνες που διέπουν αυτήν την εκχώρηση. Συντακτικά, οποιαδήποτε έκφραση μπορεί να χρησιμοποιηθεί για να αναπαραστήσει ένα όρισμα” η αξιολογούμενη τιμή εκχωρείται σε μια τοπική μεταβλητή.

Βλ. επίσης την εγγραφή του γλωσσαρίου για το *parameter*, την FAQ ερώτηση στο η διαφορά μεταξύ ορισμάτων και παραμέτρων, και **PEP 362**.

ασύγχρονος διαχειριστής context An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by **PEP 492**.

ασύγχρονος generator Μια συνάρτηση που επιστρέφει έναν *asynchronous generator iterator*. Μοιάζει με μια συνάρτηση coroutine που ορίζεται με `async def` εκτός από ότι περιέχει εκφράσεις `yield` για την παραγωγή μιας σειράς τιμών που μπορούν να χρησιμοποιηθούν σε έναν `async for` βρόχο.

Συνήθως αναφέρεται σε μια συνάρτηση ασύγχρονου generator, αλλά μπορεί να αναφέρεται σε έναν *ασύγχρονο generator iterator* σε ορισμένα contexts. Σε περιπτώσεις όπου το επιδιωκόμενο νόημα δεν είναι σαφές, με την χρήση των πλήρων όρων αποφεύγεται η ασάφεια.

Μια συνάρτηση ασύγχρονου generator μπορεί να περιέχει εκφράσεις `await` , καθώς και δηλώσεις `async for`, και `async with`.

ασύγχρονος generator iterator Ένα αντικείμενο που δημιουργήθηκε από μια συνάρτηση *asynchronous generator*.

This is an *asynchronous iterator* which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

ασύγχρονος iterable An object, that can be used in an `async for` statement. Must return an *asynchronous iterator* from its `__aiter__()` method. Introduced by **PEP 492**.

ασύγχρονος iterator An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__` must return an *awaitable* object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by **PEP 492**.

χαρακτηριστικό A value associated with an object which is referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

awaitable An object that can be used in an `await` expression. Can be a *coroutine* or an object with an `__await__()` method. See also **PEP 492**.

BDFL Ακρωνύμιο του *Benevolent Dictator For Life*, καλοκάγαθος δικτάτορας της ζωής, δηλαδή Guido van Rossum, ο δημιουργός της Python.

δυαδικό αρχείο A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

Βλ. επίσης *text file* για ένα αντικείμενο τύπου αρχείο ικανό να διαβάσει και να γράψει `str` αντικείμενα.

bytes-like αντικείμενα Ένα αντικείμενο που υποστηρίζει το bufferobjects και μπορεί να εξάγει ένα C-*contiguous* buffer. Αυτό περιλαμβάνει όλα τα αντικείμενα `bytes`, `bytearray`, και `array.array`, καθώς και πολλά κοινά `memoryview` αντικείμενα. Τα δυαδικού τύπου (bytes-like) αντικείμενα μπορούν να χρησιμοποιηθούν για διάφορες λειτουργίες που διαχειρίζονται δυαδικά δεδομένα" αυτά περιλαμβάνουν συμπίεση αποθήκευση σε δυαδικό αρχείο και αποστολή μέσω socket.

Ορισμένες λειτουργίες χρειάζονται τα δυαδικά δεδομένα να είναι μεταβλητά. Η τεκμηρίωση συχνά αναφέρεται σε αυτά ως «δυαδικά αντικείμενα ανάγνωσης-εγγραφής» (read-write bytes-like objects). Παραδείγματα μεταβλητών αντικειμένων προσωρινής αποθήκευσης περιέχουν `bytearray` και ένα `memoryview` ενός `bytearray`. Άλλες λειτουργίες απαιτούν την αποθήκευσης των δυαδικών δεδομένα σε αμετάβλητα αντικείμενα («δυαδικά αντικείμενα μόνο ανάγνωσης»" (read-only bytes-like objects) παραδείγματα αυτών περιέχουν `bytes` και ένα `memoryview` ενός `bytes` αντικειμένου.

bytecode Ο πηγαίος κώδικα της Python μεταγλωττίζεται σε *bytecode*, η εσωτερική αναπαράσταση ενός προγράμματος Python στον διερμηνέα CPython. Το *bytecode* αποθηκεύεται επίσης προσωρινά ως `.pyc` αρχεία ώστε η εκτέλεση του ίδιου αρχείου να είναι γρηγορότερη την δεύτερη φορά εκτέλεσης (μπορεί να αποφευχθεί η εκ νέου μεταγλώττιση από τον πηγαίο κώδικα σε *bytcode*). Αυτή η «ενδιάμεση γλώσσα» λέγεται ότι τρέχει σε μια *virtual machine* που εκτελεί τον κώδικα μηχανής που αντιστοιχεί σε κάθε *bytecode*. Λάβετε υπόψη ότι τα *bytecode* δεν αναμένεται να λειτουργούν μεταξύ διαφορετικών εικονικών μηχανών Python, ούτε να είναι σταθερά μεταξύ των εκδόσεων της Python.

Μια λίστα από οδηγίες σχετικά με τα bytecode μπορεί να βρεθεί στην τεκμηρίωση για το module dis.

callback Μια subroutine συνάρτηση η οποία μεταβιβάζεται ως όρισμα που θα εκτελεστεί κάποια στιγμή στο μέλλον.

κλάση Ένα πρότυπο για τη δημιουργία αντικειμένων που ορίζονται από το χρήστη. Οι ορισμοί κλάσεων συνήθως περιέχουν ορισμούς μεθόδων που λειτουργούν σε στιγμιότυπα της κλάσης.

μεταβλητή κλάσης Μια μεταβλητή που ορίζεται σε μια κλάση και προορίζεται να τροποποιηθεί μόνο σε επίπεδο κλάσης (δηλ. όχι σε ένα στιγμιότυπο μιας κλάσης).

coercion The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

μιγαδικός αριθμός Μια επέκταση του γνωστού συστήματος πραγματικών αριθμών στο οποίο όλοι οι αριθμοί εκφράζονται ως άθροισμα ενός πραγματικού μέρους και ενός φανταστικού μέρους. Οι φανταστικοί αριθμοί είναι πραγματικά πολλαπλάσια της φανταστικής μονάδα (η τετραγωνική ρίζα του `-1`), που συχνά γράφονται `i` στα μαθηματικά ή `j` στη μηχανική. Η Python έχει ενσωματωμένη υποστήριξη για μιγαδικούς αριθμούς, οι οποίοι γράφονται με αυτόν τον τελευταίο συμβολισμό" το φανταστικό μέρος γράφεται με το επίθημα `j`, π.χ., `3+1j`. Για να αποκτήσετε πρόσβαση σε σύνθετα ισοδύναμα το module `math`, χρησιμοποιήστε το `cmath`. Η χρήση μιγαδικών αριθμών είναι ένα αρκετά προηγμένο μαθηματικό χαρακτηριστικό. εάν δεν γνωρίζετε την ανάγκη τους, είναι σχεδόν σίγουρο ότι μπορείτε να τα αγνοήσετε με ασφάλεια.

διαχειριστής context An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See **PEP 343**.

context μεταβλητή Μια μεταβλητή που μπορεί να έχει πολλές διαφορετικές τιμές ανάλογα με το context. Αυτό είναι κοινό στο Thread-Local Storage όπου κάθε εκτέλεση του νήματος μπορεί να έχει διαφορετική τιμή για μια μεταβλητή. Παρόλα αυτά, με τις context μεταβλητές, μπορεί να υπάρχουν πολλά περιβάλλοντα σε ένα νήμα εκτέλεσης και η κύρια χρήση για τις context μεταβλητές είναι η παρακολούθηση των μεταβλητών σε ταυτόχρονες διεργασίες. Βλ. `contextvars`.

contiguous Ένα buffer θεωρείται contiguous ακριβώς εάν είναι είτε *C-contiguous* είτε *Fortran contriguous*. Το buffer μηδενικών διαστάσεων είναι C και Fortran contiguous. Σε μονοδιάστατους πίνακες, τα στοιχεία πρέπει να τοποθετούνται στη μνήμη το ένα δίπλα στο άλλο, με σειρά αύξησης των δεικτών ξεκινώντας από το μηδέν. Σε πολυδιάστατους C-contiguous πίνακες, ο τελευταίος δείκτης μεταβάλλεται ταχύτερα όταν επισκέπτονται τα στοιχεία σε σειρά διεύθυνσης μνήμης. Ωστόσο, σε Fortran contiguous πίνακες, ο πρώτος δείκτης μεταβάλλεται πιο γρήγορα.

coroutine Οι coroutines είναι μια πιο γενικευμένη μορφή subroutines. Οι subroutines εισάγονται σε ένα σημείο και εξάγονται σε άλλο σημείο. Οι coroutines μπορεί να εισαχθούν, να εξαχθούν και να συνεχιστούν σε πολλά διαφορετικά σημεία. Μπορούν να υλοποιήσουν με την δήλωση `async def`. Βλ. επίσης **PEP 492**.

coroutine συνάρτηση Μια συνάρτηση που επιστρέφει ένα *coroutine* αντικείμενο. Μια συνάρτηση coroutine μπορεί να ορίζεται από τη δήλωση `async def`, και μπορεί να περιέχει `await`, `async for`, και `async with` λέξεις κλειδιά. Αυτές εισήχθησαν από το **PEP 492**.

CPython Η κανονική υλοποίηση της γλώσσας προγραμματισμού Python, όπως διανέμεται στο python.org. Ο όρος «CPython» χρησιμοποιείται όταν είναι απαραίτητο για την διάκριση αυτής της υλοποίησης από άλλες όπως η *Jython* ή η *IronPython*.

decorator Μια συνάρτηση που επιστρέφει μια άλλη συνάρτηση, συνήθως εφαρμόζεται ως μετασχηματισμός συνάρτησης χρησιμοποιώντας την `@wrapper` σύνταξη. Συνηθισμένα παραδείγματα για τους decorators είναι `classmethod()` και `staticmethod()`.

Η σύνταξη του decorator είναι απλώς καλλωπιστική, οι ακόλουθοι δύο ορισμοί συναρτήσεων είναι σημασιολογικά ισοδύναμοι:

```
def f(arg):
    ...
f = staticmethod(f)

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

Η ίδια έννοια υπάρχει για τις κλάσεις, αλλά χρησιμοποιείται λιγότερο συχνά εκεί. Βλ. την τεκμηρίωση για *function definitions* και *class definitions* για περισσότερα σχετικά με τους decorators.

descriptor Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Για περισσότερες πληροφορίες αναφορικά με τις μεθόδους των descriptors, βλ. see *Implementing Descriptors* ή το Πρακτικός οδηγός για τη χρήση του Descriptor.

λεξικό An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

κατανόηση λεξικού Ένα συμπαγής τρόπος για να επεξεργαστείτε όλα ή μέρος των στοιχείων σε ένα επαναληπτικό και να επιστραφεί ένα με λεξικό με τα αποτελέσματα. `results = {n: n ** 2 for n in range(10)}` δημιουργεί ένα λεξικό που περιέχει το κλειδί `n` που αντιστοιχίζεται με την τιμή `n ** 2`. Βλ. *Displays for lists, sets and dictionaries*.

όψη λεξικού Τα αντικείμενα που επιστρέφονται από `dict.keys()`, `dict.values()`, και `dict.items()` καλούνται όψεις λεξικού. Αυτές παρέχουν μια δυναμική όψη των των εγγραφών του λεξικού, που σημαίνει ότι όταν το λεξικό μεταβάλλεται, η όψη αντικατοπτρίζει αυτές τις αλλαγές. Για να αναγκάσετε την όψη λεξικού να γίνει μια πλήρης λίστα χρησιμοποιήστε το `list(dictview)`. Βλ. dict-views.

docstring A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing Ένα στυλ προγραμματισμού που δεν εξετάζει τον τύπο ενός αντικειμένου για να προσδιορίσει αν έχει τη σωστή διεπαφή" αντίθετα, η μέθοδος ή το χαρακτηριστικό καλείται απλώς ή χρησιμοποιείται («If it looks like a duck and quacks like a duck, it must be a duck.») Δίνοντας έμφαση στις διεπαφές και όχι σε συγκεκριμένους τύπους, ο καλά σχεδιασμένος κώδικας βελτιώνει την ευελιξία του επιτρέποντας την πολυμορφική υποκατάσταση. Ο τύπος duck-typing αποφεύγει δοκιμές χρησιμοποιώντας `type()` ή `isinstance()`. (Σημείωση, ωστόσο, ότι ο τύπος πάπιας *duck-typing* μπορεί να συμπληρωθεί με *abstract base classes*.) Αντί αυτού, συνήθως χρησιμοποιεί δοκιμές `hasattr()` ή προγραμματισμό *EAFP*.

EAFP Πιο εύκολο να ζητήσεις συγχώρεση παρά άδεια. Αυτό το κοινό στυλ προγραμματισμού σε Python προϋποθέτει την ύπαρξη έγκυρων κλειδιών ή χαρακτηριστικών και συλλαμβάνει εξαιρέσεις εάν η υπόθεση αποδεχθεί εσφαλμένη. Αυτό το καθαρό και γρήγορο στυλ χαρακτηρίζεται από την παρουσία πολλών δηλώσεων *`try`* και *`except`*. Η τεχνική έρχεται σε αντίθεση με το στυλ που είναι *LBYL* κοινό σε πολλές άλλες γλώσσες, όπως η C.

έκφραση Ένα κομμάτι σύνταξης που μπορεί να αξιολογηθεί σε κάποια τιμή. Με άλλα λόγια, μια έκφραση είναι μια συσσώρευση στοιχείων έκφρασης όπως κυριολεξία, ονόματα, πρόσβαση χαρακτηριστικών, τελεστές ή κλήσεις συναρτήσεων που όλες επιστρέφουν μια τιμή. Σε αντίθεση με πολλές άλλες γλώσσες, δεν είναι όλες οι γλωσσικές δομές εκφράσεις. Υπάρχουνε επίσης *statement*s που δεν μπορούν να χρησιμοποιηθούν ως εκφράσεις, όπως το *`while`*. Οι αναθέσεις τιμών είναι επίσης δηλώσεις όχι εκφράσεις.

module επέκτασης Ένα module γραμμένο σε C ή C++, που χρησιμοποιείται από το C API της Python για να αλληλεπιδράσουν με τον πυρήνα και με τον κώδικα του χρήστη.

f-string Οι κυριολεκτικές συμβολοσειρές χρησιμοποιούν με πρόθεμα `'f'` ή `'F'` ονομάζονται συνήθως «f-strings» που είναι συντομογραφία του *formatted string literals*. Βλ. επίσης **PEP 498**.

αντικείμενο αρχείου An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

Στην πραγματικότητα υπάρχουν τρεις κατηγορίες αντικειμένων αρχείου raw *δυαδικά αρχεία*, buffered *δυαδικά αρχεία* και *αρχεία κειμένου*. Οι διεπαφές τους ορίζονται στην ενότητα `io`. Ο κανονικός τρόπος για να δημιουργήσετε ένα αντικείμενο αρχείου είναι χρησιμοποιώντας την συνάρτηση `open()`.

αντικείμενο που μοιάζει με αρχείο Ένα συνώνυμο με το *file object*.

finder Ένα αντικείμενο που προσπαθεί να βρει το *loader* για ένα module που εισήχθη.

Since Python 3.3, there are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See **PEP 302**, **PEP 420** and **PEP 451** for much more detail.

ακέραια διαίρεση Η μαθηματική διαίρεση που στρογγυλοποιεί προς τα κάτω στον κοντινότερο ακέραιο. Ο τελεστής ακέραιας διαίρεσης είναι `//`. Για παράδειγμα, η έκφραση `11 // 4` αξιολογείται σε `2` σε αντίθεση με την τιμή `2.75` που επιστρέφεται από την διαίρεση με υποδιαστολή. Σημείωση ότι `(-11) // 4` κάνει `-3` επειδή αυτή είναι η στρογγυλοποίηση *προς τα κάτω* του `-2.75`. Βλ. **PEP 238**.

συνάρτηση Μια σειρά από δηλώσεις που επιστρέφουν κάποια τιμή σε αυτόν που την κάλεσε. Σε αυτές μπορούν να περαστούν κανένα ή περισσότερα *ορίσματα* που μπορεί να χρησιμοποιηθεί για την εκτέλεση. Βλ. επίσης τις ενότητες *parameter*, *method*, και the *Function definitions*.

συνάρτηση annotation Ένας *annotation* μιας παραμέτρου συνάρτησης ή μιας τιμής επιστροφής.

Οι συναρτήσεις annotations συχνά χρησιμοποιούνται για *υποδείξεις τύπου*: για παράδειγμα, αυτή η συνάρτηση αναμένεται να πάρει δύο ορίσματα `int` και επίσης αναμένεται να έχει μία επιστρεφόμενη τιμή `int`:

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

Η σύνταξη συνάρτησης annotation αναλύεται στην ενότητα *Function definitions*.

See *variable annotation* and **PEP 484**, which describe this functionality.

__future__ Ένα *future statement*, `from __future__ import <feature>`, καθοδηγεί τον μεταγλωττιστή να μεταγλωττίσει το τρέχον module χρησιμοποιώντας σύνταξη ή σημασιολογία που θα γίνει η τυπική σε μελλοντική έκδοση της Python. Το module `__future__` τεκμηριώνει τις πιθανές τιμές του *feature*. Με την εισαγωγή αυτής της λειτουργικής μονάδας και την αξιολόγηση των μεταβλητών της, μπορείτε να δείτε πότε μια νέα δυνατότητα προστέθηκε για πρώτη φορά στην γλώσσα και πότε θα γίνει (ή έγινε) η προεπιλογή:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

συλλογή απορριμάτων Η διαδικασία απελευθέρωσης της μνήμης όταν δεν χρησιμοποιείται άλλο. Η Python εκτελεί συλλογή απορριμάτων μέσω καταμέτρησης αναφορών και ενός κυκλικού συλλέκτη σκουπιδιών που είναι σε θέση να ανιχνεύει και να σπάει τους κύκλους αναφοράς. Ο συλλέκτης απορριμάτων μπορεί να ελεγχθεί χρησιμοποιώντας το module `gc`.

generator Μια συνάρτηση που επιστρέφει ένα *generator iterator*. Μοιάζει με μια κανονική συνάρτηση εκτός από το ότι περιέχει εκφράσεις `yield` για την παραγωγή μιας σειράς τιμών που μπορούν να χρησιμοποιηθούν σε έναν βρόχο *for* ή που μπορούν να ανακτηθούν μία τη φορά με την συνάρτηση `next()` function.

Συνήθως αναφέρεται σε μια συνάρτηση generator, αλλά μπορεί να αναφέρεται σε έναν *generator iterator* σε μερικά contexts. Σε περιπτώσεις όπου το επιδιωκόμενο νόημα δεν είναι σαφές, η χρήση των πλήρων όρων αποφεύγει την ασάφεια.

generator iterator Ένα αντικείμενο που δημιουργείται από μια συνάρτηση *generator*.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator έκφραση Μια έκφραση που επιστρέφει έναν iterator. Μοιάζει με κανονική έκφραση που ακολουθείται από μια πρόταση `for` που ορίζει μια μεταβλητή βρόχου, ένα εύρος και μια προαιρετική πρόταση `if`. Η συνδυασμένη έκφραση δημιουργεί τιμές για μια συνάρτηση εγκλεισμού:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

γενική συνάρτηση Μια συνάρτηση που αποτελείται από πολλαπλές συναρτήσεις που υλοποιούν την ίδια λειτουργία για διαφορετικούς τύπους. Ποια υλοποίηση πρέπει να χρησιμοποιηθεί κατά τη διάρκεια μια κλήσης καθορίζεται από τον αλγόριθμο αποστολής.

Βλ. επίσης την καταχώρηση του *single dispatch*, τον decorator `functools.singledispatch()` και **PEP 443**.

γενικός τύπος Ένας *type* που μπορεί να παραμετροποιηθεί" συνήθως μια *container class*, όπως `list` ή `dict`. Χρησιμοποιείται για *type hints* και *annotations*.

Για περισσότερες λεπτομέρειες, βλ. generic alias types **PEP 483**, **PEP 484**, **PEP 585**, και το module `typing`.

GIL Βλ. *global interpreter lock*.

global interpreter lock Ο μηχανισμός που χρησιμοποιείται από τον διερμηνέα *CPython* για να διασφαλίσει ότι μόνο ένα νήμα εκτελεί Python *bytecode* κάθε φορά. Αυτό απλοποιεί την υλοποίηση CPython δημιουργώντας το μοντέλο αντικειμένου (συμπεριλαμβανομένων κρίσιμων ενσωματωμένων τύπων όπως π.χ. `dict`) έμμεσα ασφαλές έναντι ταυτόχρονης πρόσβασης. Το κλείδωμα ολόκληρου του διερμηνέα διευκολύνει τον διερμηνέα να είναι πολλαπλών νημάτων, εις βάρος του μεγάλου μέρους του παραλληλισμού που παρέχουν οι μηχανές πολλαπλών επεξεργαστών.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Προηγούμενες προσπάθειες να δημιουργηθεί ένας διερμηνέας «ελεύθερων-νημάτων» (αυτός που κλειδώνει τα κοινόχρηστα δεδομένα με πολύ πιο λεπτομερή ευαισθησία) δεν ήταν επιτυχείς επειδή η απόδοση υποχώρησε στην κοινή περίπτωση ενός επεξεργαστή. Πιστεύεται ότι η υπέρβαση αυτού του προβλήματος απόδοσης θα κάνουν πολύ πιο περίπλοκη και επομένως πιο δαπανηρή στην συντήρηση.

hash-based pyc Ένα αρχείο κρυφής μνήμης *bytecode* που χρησιμοποιεί τον κατακερματισμό και όχι τον χρόνο τροποποίησης του αντίστοιχου αρχείου προέλευσης για να προσδιορίσει την εγκυρότητα του. Βλ. *Cached bytecode invalidation*.

hashable An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Η ύπαρξη *hashable* κάνει ένα αντικείμενο να μπορεί να χρησιμοποιηθεί ως κλειδί λεξικού και ως μέλος ενός συνόλου, επειδή αυτές οι δομές δεδομένων χρησιμοποιούν τιμές κατακερματισμού.

Τα περισσότερα από τα αμετάβλητα ενσωματωμένα αντικείμενα της Python μπορούν να κατακερματιστούν" τα μεταβλητά κοντέινερ (όπως οι λίστες ή τα λεξικά) δεν είναι" τα αμετάβλητα κοντέινερ (όπως πλειάδες και τα frozesets) μπορούν να κατακερματιστούν μόνο εάν τα στοιχεία τους είναι κατακερματισμένα. Τα αντικείμενα που είναι στιγμιότυπα κλάσεων που ορίζονται από το χρήστη μπορούν να κατακερματιστούν από προεπιλογή. Όλα συγκρίνονται άνισα εκτός από τον εαυτό τους) και η τιμή κατακερματισμού τους προέρχεται από το `id()`.

IDLE An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

immutable Ένα αντικείμενο με σταθερή τιμή. Τα αμετάβλητα αντικείμενα περιλαμβάνουν αριθμούς , συμβολοσειρές και πλειάδες. Ένα τέτοιο αντικείμενο δεν μπορεί να αλλάξει. Ένα νέο αντικείμενο πρέπει να δημιουργηθεί εάν πρέπει να αποθηκευτεί μια διαφορετική τιμή. Παίζουν σημαντικό ρόλο σε μέρη όπου μια σταθερά απαιτείται, για παράδειγμα ως κλειδί σε ένα λεξικό.

εισαγόμενο path Μια λίστα από τοποθεσίες (ή *καταχωρίσεις διαδρομής*) που μπορούν να αναζητηθούν *path based finder* για να εισαχθούν modules. Κατά την διαδικασία εισαγωγής, αυτή η λίστα με τοποθεσίες συνήθως έρχεται από `sys.path`, αλλά για τα υποπακέτα μπορεί επίσης να έρθει από το χαρακτηριστικό του πακέτου γονέα `__path__`.

εισαγωγή Η διαδικασία κατά την οποία ο κώδικας της Python σε ένα module είναι διαθέσιμη στον κώδικα Python ενός άλλου module.

εισαγωγέας Ένα αντικείμενο μπορεί και να αναζητεί και να φορτώνει ένα module" και ένα *finder* και *loader* αντικείμενο.

διαδραστικός Η Python έχει έναν διαδραστικό διερμηνέα όπου σημαίνει ότι μπορείς να εισάγεις δηλώσεις και εκφράσεις στην εισαγωγή εντολών του διερμηνέα, εκτελώντας τες άμεσα και εμφανίζοντας τα αντικείμενα. Απλώς εκκινήστε την `python` χωρίς ορίσματα (πιθανώς επιλέγοντας το από το κύριο μενού του υπολογιστή σας). Αποτελεί έναν αποδοτικό τρόπο για να δοκιμάστε νέες ιδέες ή να εξετάστε λειτουργικές μονάδες και πακέτα (θυμηθείτε `help(x)`).

interpreted Η Python είναι μια interpreted γλώσσα, σε αντίθεση με μια μεταγλωττισμένη, αν και η διάκριση μπορεί να είναι και θολή λόγω της παρουσία του bytecode μεταγλωττιστή. Αυτό σημαίνει ότι τα αρχεία προέλευσης μπορούν να εκτελεστούν απευθείας χωρίς να δημιουργηθεί ρητά ένα εκτελέσιμο αρχείο που στην συνέχεια εκτελείται. Οι interpreted γλώσσες συνήθως έχουν μικρότερο κύκλο ανάπτυξης/ εντοπισμού σφαλμάτων από τις μεταγλωττισμένες, αν και τα προγράμματά τους γενικά εκτελούνται πιο αργά. Βλ. επίσης *interactive*.

τερματισμός λειτουργίας διερμηνέα Όταν ζητείται τερματισμός λειτουργίας, ο διερμηνέας της Python εισέρχεται σε μια ειδική φάση όπου απελευθερώνει σταδιακά όλους τους διατιθέμενους πόρους, όπως λειτουργικές μονάδες και πολλαπλές κρίσιμες εσωτερικές δομές. Επίσης πραγματοποιεί αρκετές κλήσεις στο *συλλέκτης σκουπιδιών*. Αυτό μπορεί να ενεργοποιήσει την εκτέλεση κώδικα σε καταστροφείς που ορίζονται από το χρήστη ή σε callbacks ασθενούς ανταποκρίσεις. Ο κώδικας που εκτελείται κατά τη φάση τερματισμού λειτουργίας μπορεί να συναντήσει διάφορες εξαιρέσεις, καθώς οι πόροι στους οποίους βασίζεται ενδέχεται να μην λειτουργούν πλέον (συνήθη παραδείγματα είναι οι λειτουργικές μονάδες βιβλιοθήκης ή ο μηχανισμός ειδοποιήσεων).

Ο βασικός λόγος τερματισμού λειτουργίας του διερμηνέα είναι ότι το `__main__` module ή ολοκληρώθηκε η εκτέλεση του κώδικα που έτρεχε.

iterable An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *Sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Περισσότερες πληροφορίες μπορούν να βρεθούν στο typeiter.

συνάρτηση key Μια συνάρτηση κλειδί ή μια συνάρτηση ταξινόμησης είναι μια δυνατότητα κλήσης που επιστρέφει μια τιμή που χρησιμοποιείται για ταξινόμηση ή διάταξη. Για παράδειγμα, `locale.strxfrm()` χρησιμοποιείται για την παραγωγή ενός κλειδιού ταξινόμησης που γνωρίζει τις συμβάσεις ταξινόμησης για συγκεκριμένες τοπικές ρυθμίσεις.

Ένα αριθμός εργαλείων στην Python δέχεται βασικές συναρτήσεις για τον έλεγχο του τρόπου με τον οποίο τα στοιχεία ταξινομούνται ή ομαδοποιούνται. Αυτά περιέχουν `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq. nlargest()`, και `itertools. groupby()`.

There are several ways to create a key function. For example. the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the Sorting HOW TO for examples of how to create and use key functions.

όρισμα keyword Βλ. *argument*.

lambda Μια ανώνυμη ενσωματωμένη συνάρτηση που αποτελείται από μια μοναδική *expression* η οποία αξιολογείται όταν καλείται η συνάρτηση. Η σύνταξη για τη δημιουργία μιας συνάρτησης lambda είναι `lambda [parameters]: expression`

LBYL Look before you leap. Αυτό το στυλ κωδικοποίησης ελέγχει ρητά τις προϋποθέσεις πριν πραγματοποιήσει κλήσεις ή αναζητήσεις. Αυτό το στυλ έρχεται σε αντίθεση με την προσέγγιση *EAFP* και χαρακτηρίζεται από την παρουσία πολλών δηλώσεων `if`.

Σε ένα περιβάλλον πολλαπλών νημάτων, η προσέγγιση LBYL μπορεί να διακινδυνεύσει να εισάγει μια συνθήκη αγώνα μεταξύ «the Looking» και «the leaping». Για παράδειγμα ο κώδικας, `if key in mapping: return mapping[key]` μπορεί να αποτύχει εάν ένα άλλο νήμα αφαιρέσει το *key* από το *mapping* μετά τη δοκιμή, αλλά πριν από την αναζήτηση. Αυτό το πρόβλημα μπορεί να λυθεί με κλειδώματα ή χρησιμοποιώντας την προσέγγιση EAFP.

λίστα A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is O(1).

list comprehension Ένα συμπαγής τρόπος για να επεξεργαστείτε όλα ή μέρος των στοιχείων σε μια ακολουθία και να επιστρέψετε μια λίστα με τα αποτελέσματα. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` δημιουργεί μια λίστα συμβολοσειρών που περιέχουν ζυγούς δεκαεξαδικούς αριθμούς (0x..) στο εύρος από 0 έως 255. Η πρόταση `if` είναι προαιρετική. Εάν παραλειφθεί, όλα τα στοιχεία στο `range(256)` υποβάλλονται σε επεξεργασία.

loader An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details and `importlib.abc.Loader` for an *abstract base class*.

μαγική μέθοδος Ένα άτυπο συνώνυμο για *special method*.

mapping A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

meta path finder Ένας *finder* που επιστράφηκε με αναζήτηση στο `sys.meta_path`. Οι finders μετα-διαδρομής σχετίζονται, αλλά διαφέρουν από τα *finders entry διαδρομής*.

Βλ. `importlib.abc.MetaPathFinder` για τις μεθόδους που υλοποιούν οι meta path finders.

μετα-κλάση Η κλάση μιας κλάσης. Οι ορισμοί κλάσης δημιουργούν ένα όνομα κλάσης, ένα λεξικό κλάσης και μια λίστα βασικών κλάσεων. Η μετα-κλάση είναι υπεύθυνη για την απόκτηση αυτών των τριών ορισμάτων και την δημιουργία της κλάσης. Οι περισσότερες αντικειμενοστρεφείς γλώσσες προγραμματισμού παρέχουν μια προεπιλεγμένη υλοποίηση. Αυτό που κάνει την Python ξεχωριστή είναι ότι είναι δυνατή η δημιουργία προσαρμοσμένων μετακλάσεων. Οι περισσότεροι χρήστες δεν χρειάζονται ποτέ αυτό το εργαλείο, αλλά όταν παραστεί ανάγκη, αυτό το εργαλείο, οι μετα-κλάσεις μπορούν να παρέχουν ισχυρές, κομψές λύσεις. Έχουν χρησιμοποιηθεί για την καταγραφή πρόσβασης χαρακτηριστικών, την προσθήκη ασφάλειας νημάτων, την παρακολούθηση δημιουργίας αντικειμένων, την υλοποίηση *singletons*, και πολλές άλλες εργασίες.

Περισσότερες πληροφορίες μπορούν να βρεθούν στο *Metaclasses*.

μέθοδος Μια συνάρτηση που ορίζεται μέσα στο σώμα μιας κλάσης. Εάν καλείται ως χαρακτηριστικό μιας περίπτωσης αυτής της κλάσης, η μέθοδος θα λάβει αντικείμενο περίπτωσης ως πρώτο της *argument* (το οποίο συνήθως ονομάζεται `self`). Βλ. *function* και *nested scope*.

σειρά ανάλυσης μεθόδων Method Resolution Order is the order in which base classes are searched for a member during lookup. See The Python 2.3 Method Resolution Order for details of the algorithm used by the Python interpreter since the 2.3 release.

module Ένα αντικείμενο που χρησιμεύει ως οργανωτική μονάδα του κώδικα της Python. Τα modules έχουν έναν χώρο ονομάτων που περιέχει αυθαίρετα αντικείμενα Python. Τα modules φορτώνονται στην Python με την διαδικασία *importing*.

Βλ. επίσης *package*.

τεχνικές προδιαγραφές module Ένα namespace που περιέχει τις πληροφορίες που σχετίζονται με την εισαγωγή που χρησιμοποιούνται για την φόρτωση ενός module. Μια περίπτωση του `importlib.machinery.ModuleSpec`.

MRO Βλ. *method resolution order*.

mutable Τα ευμετάβλητα αντικείμενα μπορούν να αλλάξουν τις τιμές αλλά να κρατήσουν τα `id()`. Βλ. επίσης *immutable*.

named tuple Ο όρος «named tuple» εφαρμόζεται για οποιονδήποτε τύπο ή κλάση που κληρονομείται από την πλειάδα και των οποίων τα στοιχεία μπορούν να ευρετηριοποιηθούν είναι προσβάσιμα χρησιμοποιώντας επώνυμα χαρακτηριστικά. Ο τύπος ή η κλάση μπορεί να έχει και άλλα χαρακτηριστικά.

Πολλοί ενσωματωμένοι τύποι είναι named tuples, συμπεριλαμβανομένων των τιμών που επιστρέφονται από `time.localtime()` και `os.stat()`. Ένα άλλο παράδειγμα είναι το `sys.float_info`:

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand or it can be created with the factory function `collections.namedtuple()`. The latter technique also adds some extra methods that may not be found in hand-written or built-in named tuples.

namespace Το μέρος όπου αποθηκεύεται μια μεταβλητή. Τα namespaces υλοποιούνται ως λεξικά. Υπάρχουν οι τοπικοί, οι καθολικοί και οι ενσωματωμένοι namespaces καθώς και οι ένθετοι namespaces σε αντικείμενα (σε μεθόδους). Για παράδειγμα οι συναρτήσεις `builtins.open` και `os.open()` διακρίνονται από τους χώρους ονομάτων τους. Οι χώροι ονομάτων βοηθούν επίσης την αναγνωσιμότητα και τη συντηρησιμότητα καθιστώντας σαφές ποιο module υλοποιεί μια λειτουργία. Για παράδειγμα, γράφοντας `random.seed()` ή `itertools.islice()` καθιστά σαφές ότι αυτές οι συναρτήσεις υλοποιούνται από τα module `random` και `itertools`, αντίστοιχα.

πακέτο namespace A **PEP 420** *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.

Βλ. επίσης *module*.

nested scope Η δυνατότητα αναφοράς σε μια μεταβλητή σε έναν περικλειόμενο ορισμό. Για παράδειγμα μια συνάρτηση που ορίζεται μέσα σε μια άλλη συνάρτηση μπορεί να αναφέρεται σε μεταβλητές στην εξωτερική συνάρτηση. Σημειώστε ότι τα ένθετα πεδία από προεπιλογή λειτουργούν μόνο για αναφορά και όχι για εκχώρηση. Οι τοπικές μεταβλητές διαβάζονται και γράφονται στο εσωτερικό πεδίο εφαρμογής. Ομοίως, οι καθολικές μεταβλητές διαβάζουν και γράφουν στον καθολικό χώρο ονομάτων. Το `nonlocal` επιτρέπει την εγγραφή σε εξωτερικά πεδία.

κλάση νέου στυλ Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

αντικείμενο Οποιαδήποτε δεδομένα με κατάσταση (χαρακτηριστικά ή τιμή) και καθορισμένη συμπεριφορά (μέθοδοι). Επίσης, η τελική βασική κλάση οποιασδήποτε *new-style class*.

πακέτο A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__path__` attribute.

Βλ. επίσης *regular package* και *namespace package*.

παράμετρος Μια έγκυρη οντότητα σε έναν ορισμό *function* (ή μέθοδος) που καθορίζει ένα *argument* (ή σε ορισμένες περιπτώσεις, ορίσματα) που μπορεί να δεχθεί η συνάρτηση. Υπάρχουν πέντε είδη παραμέτρων:

- *λέξη-κλειδί ή θέση*: καθορίζει ένα όρισμα που μπορεί να μεταβιβαστεί είτε *θέσεως* ή ως *όρισμα λέξης-κλειδιού*. Αυτό είναι το προεπιλεγμένο είδος παραμέτρου, για παράδειγμα *foo* και *bar* στα ακόλουθα:

```
def func(foo, bar=None): ...
```

- *θέσεως μόνο*: καθορίζει ένα όρισμα που μπορεί να παρέχεται μόνο από τη θέση. Οι παράμετροι μόνο θέσης μπορούν να οριστούν συμπεριλαμβάνοντας έναν χαρακτήρα / στη λίστα παραμέτρων του ορισμού συνάρτησης μετά από αυτές, για παράδειγμα *posonly1* και *posonly2* στα εξής:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *λέξης-κλειδί μόνο*: καθορίζει ένα όρισμα που μπορεί να παρέχεται μόνο με λέξη κλειδί. Οι παράμετροι μόνο για λέξη-κλειδί μπορούν να οριστούν συμπεριλαμβάνοντας μια παράμετρο θέσης ή σκέτο * στη λίστα παραμέτρων του ορισμού συνάρτησης πριν από αυτές, για παράδειγμα *kw_only1* και *kw_only2* στα ακόλουθα:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *μεταβλητή θέσης*: καθορίζει ότι μπορεί να παρασχεθεί μια αυθαίρετη ακολουθία ορισμάτων θέσης (επιπλέον των ορισμάτων θέσης που είναι ήδη αποδεκτά από άλλες παραμέτρους). Μια τέτοια παράμετρος μπορεί να οριστεί προσαρτώντας το όνομα της παραμέτρου με *, για παράδειγμα *args* στα ακόλουθα:

```
def func(*args, **kwargs): ...
```

- *μεταβλητή λέξη-κλειδί*: καθορίζει ότι μπορούν να παρέχονται αυθαίρετα πολλά ορίσματα λέξης-κλειδιού (επιπλέον των ορισμάτων λέξης κλειδιού που είναι αποδεκτά από άλλες παραμέτρους). Μια τέτοια παράμετρος μπορεί να οριστεί προσαρτώντας το όνομα της παραμέτρου με **, για παράδειγμα *kwargs* όπως παραπάνω.

Οι παράμετροι μπορούν να καθορίσουν τόσο τα προαιρετικά όσο και τα απαιτούμενα ορίσματα , καθώς και προεπιλεγμένες τιμές για ορισμένα προαιρετικά ορίσματα.

Βλ. επίσης την *argument* καταχώριση ευρετηρίου, την ερώτηση FAQ σχετικά με η διαφορά μεταξύ ορισμάτων και παραμέτρων, την κλάση `inspect.Parameter`, την ενότητα *Function definitions* και **PEP 362**.

path entry Μια μεμονωμένη τοποθεσία στο *import path* την οποία συμβουλεύεται ο *path based finder* για να βρει modules για εισαγωγή.

path entry finder Ένας *finder* που επιστρέφεται από έναν καλούμενο στο `sys.path_hooks` (δηλαδή ένα *path entry hook*) που ξέρει πως να εντοπίζει modules με *path entry*.

Βλ. `importlib.abc.PathEntryFinder` για τις μεθόδους που ο entry finder διαδρομής υλοποιεί.

path entry hook A callable on the `sys.path_hook` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

path based finder Ένα από τα προεπιλεγμένα *meta path finders* που αναζητά ένα *import path* για modules.

path-like αντικείμενο Ένα αντικείμενο που αντιπροσωπεύει ένα path συστήματος αρχείων. Ένα αντικείμενο path είναι είτε ένα αντικείμενο `str` ή `bytes` που αντιπροσωπεύει ένα path ή ένα αντικείμενο που υλοποιεί το πρωτόκολλο `os.PathLike`. Ένα αντικείμενο που υποστηρίζει το πρωτόκολλο `os.PathLike` μπορεί να μετατραπεί σε path συστήματος αρχείων `str` ή `bytes` καλώντας την συνάρτηση `os.fspath()`” τα `os.fsdecode()` και `os.fsencode()` μπορούν να χρησιμοποιηθούν για την εγγύηση ενός αποτελέσματος `str` ή `bytes`, αντίστοιχα. Εισήχθη από τον **PEP 519**.

PEP Πρόταση Βελτίωσης Python. Ένα PEP είναι ένα έγγραφο σχεδιασμού που παρέχει πληροφορίες στην κοινότητα Python ή περιγράφει μια νέα δυνατότητα για την Python ή τις διαδικασίες ή το περιβάλλον της. Τα PEP θα πρέπει να παρέχουν μια συνοπτική τεχνική προδιαγραφή και μια λογική για τα προτεινόμενα χαρακτηριστικά.

Τα PEP προορίζονται να είναι οι κύριοι μηχανισμοί για την πρόταση σημαντικών νέων χαρακτηριστικών, για τη συλλογή πληροφοριών της κοινότητας για ένα ζήτημα και για την τεκμηρίωση των αποφάσεων σχεδιασμού που έχουν εισαχθεί στην Python. Ο συγγραφέας του PEP είναι υπεύθυνος για την οικοδόμηση συναίνεσης εντός της κοινότητας και την τεκμηρίωση αντίθετων απόψεων.

Βλ. **PEP 1**.

τμήμα Ένα σύνολο από αρχεία σε έναν μόνο κατάλογο (ενδεχομένως αποθηκευμένο σε αρχείο *zip*) που συμβάλλουν σε ένα namespace πακέτο, όπως ορίζεται στο **PEP 420**.

όρισμα θέσης Βλ. *argument*.

provisional API Ένα provisional API είναι αυτό που έχει εσκεμμένα εξαιρεθεί από τις backwards εγγυήσεις συμβατότητας της τυπικής βιβλιοθήκης. Αν και δεν αναμένονται σημαντικές αλλαγές σε τέτοιες διεπαφές, εφόσον επισημαίνονται ως προσωρινές, αλλαγές μη backwards συμβατότητας (μέχρι και κατάργηση της διεπαφής) μπορεί να προκύψουν εάν κριθεί απαραίτητο από τους βασικούς προγραμματιστές. Τέτοιες αλλαγές δεν θα γίνουν άσκοπα – θα συμβούν μόνο εάν αποκαλυφθούν σοβαρά θεμελιώδη ελαττώματα που παραλείφθηκαν πριν από τη συμπερίληψη του API.

Ακόμη και για provisional API, οι μη backwards συμβατές αλλαγές θεωρούνται «λύση έσχατης ανάγκης»- θα εξακολουθεί να γίνεται κάθε προσπάθεια για να βρεθεί μια λύση backwards συμβατή σε τυχόν εντοπισμένα προβλήματα.

Αυτή η διαδικασία επιτρέπει στην τυπική βιβλιοθήκη να συνεχίσει να εξελίσσεται με την πάροδο του χρόνου, χωρίς να κλειδώνει προβληματικά σφάλματα σχεδιασμού για εκτεταμένες χρονικές περιόδους. Βλ. **PEP 411** για περισσότερες λεπτομέρειες.

provisional πακέτο Βλ. *provisional API*.

Python 3000 Ψευδώνυμο για το σύνολο εκδόσεων Python 3.x (επινοήθηκε πριν από πολύ καιρό όταν η κυκλοφορία της έκδοσης 3 ήταν κάτι στο μακρινό μέλλον.) Αυτό ονομάζεται επίσης ως συντομογραφία «Py3k».

Pythonic Μια ιδέα ή ένα κομμάτι κώδικα που ακολουθεί πιστά τα πιο κοινά ιδιώματα της γλώσσας Python, αντί να υλοποιεί κώδικα χρησιμοποιώντας έννοιες κοινές σε άλλες γλώσσες. Για παράδειγμα, ένα κοινό ιδίωμα στην Python είναι να κάνει μια επανάληψη πάνω από όλα τα στοιχεία ενός iterable χρησιμοποιώντας μια δήλωση `for`. Πολλές άλλες γλώσσες που δεν έχουν αυτόν τον τύπο κατασκευής, έτσι οι άνθρωποι που δεν είναι εξοικειωμένοι με την Python χρησιμοποιούν μερικές φορές έναν αριθμητικό μετρητή:

```
for i in range(len(food)):
    print(food[i])
```

Αντίθετα, μια πιο καθαρή μέθοδος Pythonic:

```
for piece in food:
    print(piece)
```

αναγνωρισμένο όνομα Ένα όνομα με κουκκίδες που δείχνει τη «διαδρομή» από το καθολικό εύρος ενός module σε μια κλάση, συνάρτηση ή μέθοδο που ορίζεται σε αυτήν την ενότητα, όπως ορίζεται στο **PEP 3155**. Για συναρτήσεις και κλάσεις ανώτατου επιπέδου, το αναγνωρισμένο όνομα είναι ίδιο με το όνομα του αντικειμένου:

```
>>> class C:
...     class D:
...         def meth(self):
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Όταν χρησιμοποιείται για αναφορά σε modules , το *πλήρως αναγνωρισμένο όνομα* σημαίνει ολόκληρο το διακεκομμένο path προς το module, συμπεριλαμβανομένων τυχόν γονικών πακέτων π.χ. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

πλήθος αναφοράς The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. The `sys` module defines a `getrefcount()` function that programmers can call to return the reference count for a particular object.

κανονικό πακέτο Ένα παραδοσιακό *package*, όπως ένας κατάλογος που περιέχει ένα `__init__.py` αρχείο.

Βλ. επίσης *namespace package*.

__slots__ Μια δήλωση μέσα σε μια κλάση που εξοικονομεί μνήμη δηλώνοντας εκ των προτέρων χώρο για παράδειγμα χαρακτηριστικά και εξαλείφοντας λεξικά στιγμιοτύπων. Αν και δημοφιλής, η τεχνική είναι κάπως δύσκολο να γίνει σωστή και προορίζεται καλύτερα για σπάνιες περιπτώσεις όπου υπάρχει μεγάλος αριθμός στιγμιοτύπων σε μια εφαρμογή κρίσιμης-μνήμης.

ακολουθία An *iterable* which supports efficient element access using integer indices via the *`__getitem__()`* special method and defines a *`__len__()`* method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports *`__getitem__()`* and *`__len__()`*, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just *`__getitem__()`* and *`__len__()`*, adding `count()`, `index()`, *`__contains__()`*, and *`__reversed__()`*. Types that implement this expanded interface can be registered explicitly using `register()`.

set comprehension Ένας συμπαγής τρόπος για να επεξεργαστείτε όλα ή μέρος των στοιχείων σε ένα iterable και να επιστραφεί ένα σύνολο με τα αποτελέσματα. `results = {c for c in 'abracadabra' if c not in 'abc'}` δημιουργεί το σύνολο συμβολοσειρών `{'r', 'd'}`. Βλ. *Displays for lists, sets and dictionaries*.

μοναδικό dispatch Μια μορφή dispatch *generic function* όπου η υλοποίηση επιλέγεται με βάση τον τύπο ενός μεμονωμένου ορίσματος.

slice Ένα αντικείμενο που συνήθως περιέχει ένα τμήμα μιας ακολουθίας *sequence*. Δημιουργείται ένα slice χρησιμοποιώντας τη σημείωση subscript, `[]` με άνω και κάτω τελείες μεταξύ αριθμών όταν δίνονται πολλοί, όπως στο `variable_name[1:3:5]`. Η σημείωση αγκύλης (subscript) χρησιμοποιεί εσωτερικά αντικείμενα `slice`.

ειδική μέθοδος Μια μέθοδος που καλείται σιωπηρά από την Python για να εκτελέσει μια συγκεκριμένη λειτουργία σε έναν τύπο, όπως η προσθήκη. Τέτοιες μέθοδοι έχουνε ονόματα που ξεκινούν και τελειώνουν με διπλές κάτω παύλες. Οι ειδικές μέθοδοι τεκμηριώνονται στο *Special method names*.

δήλωση Μια πρόταση είναι μέρος μιας σουίτας (ένα «μπλοκ» κώδικα). Μια πρόταση είναι είτε ένας *expression* είτε μια από πολλές δομές με μια λέξη-κλειδί όπως *if*, *while* ή *for*.

κωδικοποίηση κειμένου Μια συμβολοσειρά στην Python είναι μια ακολουθία σημείων κώδικα Unicode (στο εύρος U+0000–U+10FFFF). Για να αποθηκεύσετε ή να μεταφέρετε μια συμβολοσειρά, πρέπει να σειριοποιηθεί ως δυαδική ακολουθία.

Η σειριοποίηση μιας συμβολοσειράς σε μια δυαδική ακολουθία είναι γνωστή ως «κωδικοποίηση» , και η αναδημιουργία της συμβολοσειράς από την δυαδική ακολουθία είναι γνωστή ως «αποκωδικοποίηση».

Υπάρχει μια ποικιλία διαφορετικής σειριοποίησης κειμένου codecs, οι οποίοι συλλογικά αναφέρονται ως «κωδικοποιήσεις κειμένου».

αρχείο κειμένου Ένα *file object* ικανό να διαβάζει και να γράφει αντικείμενα `str`. Συχνά, ένα αρχείο κειμένου αποκτά πραγματικά πρόσβαση σε μια ροή δυαδική ροή δεδομένων και χειρίζεται αυτόματα την *text encoding*. Παραδείγματα αρχείων κειμένου είναι αρχεία που ανοίγουν σε λειτουργία κειμένου (`'r'` ή `'w'`), `sys.stdin`, `sys.stdout`, και στιγμιότυπα του `io.StringIO`.

Βλ. επίσης *binary file* για ένα αντικείμενο αρχείου με δυνατότητα ανάγνωσης και εγγραφής *δυαδικά αντικείμενα*.

συμβολοσειρά τριπλών εισαγωγικών Μια συμβολοσειρά που δεσμεύεται από τρεις περιπτώσεις είτε ενός εισαγωγικού (») ή μιας αποστρόφου ("). Αν και δεν παρέχουν καμία λειτουργικότητα που δεν είναι διαθέσιμη με συμβολοσειρές με μονά εισαγωγικά, είναι χρήσιμες για διαφόρους λόγους. Σας επιτρέπουν να συμπεριλάβετε μονά και διπλά εισαγωγικά χωρίς διαφυγή σε μια συμβολοσειρά και μπορούν να εκτείνονται σε πολλές γραμμές χωρίς τη χρήση του χαρακτήρα συνέχεια, καθιστώντας τα ιδιαίτερα χρήσιμα κατά τη σύνταξη εγγράφων με συμβολοσειρές.

τύπος The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

type alias Ένα συνώνυμο για έναν τύπο, που δημιουργείται με την ανάθεση τύπου σε ένα αναγνωριστικό.

Τα type aliases είναι χρήσιμα για την απλοποίηση *type alias*. Για παράδειγμα:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

μπορεί να γίνει πιο ευανάγνωστο όπως:

```
Color = tuple[int, int, int]

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

Βλ. `typing` και **PEP 484**, που περιγράφει αυτήν την λειτουργικότητα.

type hint Ένας *annotation* που καθορίζει τον αναμενόμενο τύπο για μια μεταβλητή, ένα χαρακτηριστικό κλάσης ή μια παράμετρο συνάρτησης ή τιμή επιστροφής.

Type hints are optional and are not enforced by Python but they are useful to static type analysis tools, and aid IDEs with code completion and refactoring.

Υποδείξεις τύπου (type hints) για καθολικές μεταβλητές, χαρακτηριστικά κλάσης και συναρτήσεις , αλλά όχι τοπικές μεταβλητές, μπορούν να προσπελαστούν χρησιμοποιώντας το `typing.get_type_hints()`.

Βλ. `typing` και **PEP 484**, που περιγράφει αυτήν την λειτουργικότητα.

καθολικές νέες γραμμές Ένα τρόπος ερμηνείας ροών κειμένου στον οποίο όλα τα ακόλουθα αναγνωρίζονται ως λήξεις μιας γραμμής: η σύμβαση τέλους γραμμής του Unix `'\n'`, η σύμβαση των Windows `'\r\n'`, και την παλιά σύμβαση Macintosh `'\r'`. Βλ. **PEP 278** και **PEP 3116**, καθώς και `bytes.splitlines()` για πρόσθετη χρήση.

annotation μεταβλητής Ένας *annotation* μια μεταβλητής ή ενός χαρακτηριστικού κλάσης.

Όταν annotating μια μεταβλητή ή ένα χαρακτηριστικό κλάσης, η ανάθεση είναι προαιρετική:

```
class C:
    field: 'annotation'
```

Τα annotations μεταβλητών χρησιμοποιούνται συνήθως για *type hints*: για παράδειγμα αυτή η μεταβλητή αναμένεται να λάβει τιμές `int`:

```
count: int = 0
```

Η σύνταξη annotation μεταβλητής περιγράφεται στην ενότητα *Annotated assignment statements*.

See *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality.

virtual environment Ένα συνεργατικά απομονωμένο περιβάλλον χρόνου εκτέλεσης που επιτρέπει στους χρήστες και τις εφαρμογές της Python να εγκαταστήσουν και να αναβαθμίσουν πακέτα διανομής Python χωρίς να παρεμβαίνουν στη συμπεριφορά άλλων εφαρμογών Python που εκτελούνται στο ίδιο σύστημα.

Βλ. επίσης `venv`.

virtual machine Ένας υπολογιστής ορίζεται εξ ολοκλήρου από το λογισμικό. Η εικονική μηχανή της Python εκτελεί το *bytecode* που εκπέμπεται από τον μεταγλωττιστή bytecode.

Zen της Python Κατάλογος σχεδιαστικών αρχών και φιλοσοφιών που είναι χρήσιμες για την κατανόηση και τη χρήση της γλώσσας. Ο κατάλογος μπορεί να βρεθεί πληκτρολογώντας «`import this`» στην διαδραστική κονσόλα.

ΠΑΡΑΡΤΗΜΑ B′

About these documents

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

Η ανάπτυξη των εγγράφων και των εργαλείων τους είναι εξ" ολοκλήρου εθελοντική προσπάθεια, όπως και η ίδια η Python. Εάν θέλετε να συνεισφέρετε, ρίξτε μια ματιά στη σελίδα reporting-bugs για πληροφορίες σχετικές με το πως να το κάνετε. Καινούριοι εθελοντές είναι πάντα ευπρόσδεκτοι!

Πολλές ευχαριστίες πηγαίνουν στους:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- the Docutils project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh για το δικό του Alternative Python Reference πρότζεκτ από το οποίο το Sphinx πήρε πολύ καλές ιδέες.

B′.1 Contributors to the Python Documentation

Πολλοί άνθρωποι έχουν συνεισφέρει στη γλώσσα Python, την βιβλιοθήκη της Python, και τα έγγραφα της Python. Δείτε Misc/ACKS στις πηγές διανομής της Python για μια λίστα των συντελεστών.

Μόνο με τη συμβολή και τις συνεισφορές της κοινότητας της Python, η Python έχει τέτοια υπέροχα έγγραφα – Σας ευχαριστούμε!

ΠΑΡΑΡΤΗΜΑ Γ'

Ιστορία και Άδεια

Γ'.1 Η ιστορία του λογισμικού

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

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

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

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

Έκδοση	Προερχόμενη από	Έτος	Ιδιοκτησία	GPL compatible?
0.9.0 έως 1.2	δ/υ	1991-1995	CWI	ναι
1.3 έως 1.5.2	1.2	1995-1999	CNRI	ναι
1.6	1.5.2	2000	CNRI	όχι
2.0	1.6	2000	BeOpen.com	όχι
1.6.1	1.6	2001	CNRI	όχι
2.1	2.0+1.6.1	2001	PSF	όχι
2.0.1	2.0+1.6.1	2001	PSF	ναι
2.1.1	2.1+2.0.1	2001	PSF	ναι
2.1.2	2.1.1	2002	PSF	ναι
2.1.3	2.1.2	2002	PSF	ναι
2.2 και πάνω	2.1.1	2001-σήμερα	PSF	ναι

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

Χάρη, στους πολλούς εξωτερικούς εθελοντές που εργάστηκαν κάτω από τις οδηγίες του Guido, αυτές οι εκδόσεις έγιναν εφικτές.

Γ'.2 Όροι και προϋποθέσεις για την πρόσβαση ή την χρήση της Python με άλλους τρόπους

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Κάποιο λογισμικό που είναι ενσωματωμένο στην Python είναι υπό διαφορετικές άδειες χρήσης. Οι άδειες παρατίθενται με κώδικα που εμπίπτει σε αυτήν την άδεια. Δείτε *Άδειες και Ευχαριστίες για Ενσωματωμένο Λογισμικό* για μια ελλιπή λίστα αυτών των αδειών.

Γ'.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.9.23

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.9.23 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.9.23 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.9.23 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.9.23 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   3.9.23.

4. PSF is making Python 3.9.23 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
```

```
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 3.9.23 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.9.23
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.9.23, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

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

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 3.9.23, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

Γ'.2.2 ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ BEOPEN.COM ΓΙΑ PYTHON 2.0

ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ ΑΝΟΙΧΤΟΥ ΚΩΔΙΚΑ BEOPEN PYTHON ΕΚΔΟΣΗ 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,
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
   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.
```

Γ'.2.3 ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ CNRI ΓΙΑ 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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
   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.
```

Γ'.2.4 ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ CWI ΓΙΑ PYTHON 0.9.0 ΕΩΣ 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.
```

Γ'.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.9.23 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
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.
```

Γ'.3 Άδειες και Ευχαριστίες για Ενσωματωμένο Λογισμικό

Αυτή η ενότητα είναι μια ημιτελής, αλλά αυξανόμενη λίστα αδειών και ευχαριστιών για λογισμικό τρίτων, που ενσωματώνεται στην διανομή της Python.

Γ'.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

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

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

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

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

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

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

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

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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)
```

Γ'.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

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

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

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.
```

Γ'.3.3 Ασύγχρονες socket υπηρεσίες

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

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

Γ'.3.4 Διαχείριση Cookie

Η ενότητα `http.cookies` περιέχει την παρακάτω ειδοποίηση:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

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

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

Γ'.3.5 Ανίχνευση εκτέλεσης

Η ενότητα `trace` περιέχει την παρακάτω ειδοποίηση:

Γ'.3.6 Συναρτήσεις UUencode και UUdecode

Η ενότητα `uu` περιέχει την παρακάτω ειδοποίηση:

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

Γ'.3.7 Κλήσεις Απομακρυσμένης Διαδικασίας XML

Η ενότητα `xmlrpc.client` περιέχει την παρακάτω ειδοποίηση:

```
    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.
```

Γ'.3.8 test_epoll

The `test_epoll` module contains the following notice:

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

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

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

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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.
```

Γ'.3.9 Επιλογή kqueue

Η ενότητα `select` περιέχει την παρακάτω ειδοποίηση για την kqueue διεπαφή:

```
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.
```

Γ'.3.10 SipHash24

Το αρχείο `Python/pyhash.c` περιέχει την υλοποίηση του Marek Majkowski του αλγορίθμου τού Dan Bernstein, SipHash24. Αυτό περιέχει την παρακάτω σημείωση:

```
<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>
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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)
```

Γ'.3.11 strtod και dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

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

Γ'.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

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

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

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

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

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

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

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

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

Γ'.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

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

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

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

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

Γ'.3.14 libffi

The `_ctypes` extension is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

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

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

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

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

Γ'.3.15 zlib

Η επέκταση `zlib` δημιουργείται χρησιμοποιώντας ένα συμπεριλαμβανόμενου αντίγραφο των πηγών zlib, εάν η έκδοση του zlib που βρίσκεται στο σύστημα είναι πολύ παλιά για να χρησιμοποιηθεί για την κατασκευή:

```
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
```

Γ'.3.16 cfuhash

Η υλοποίηση του πίνακα κατακερματισμού που χρησιμοποιείται από το `tracemalloc` βασίζεται στο έργο cfuhash:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

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

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

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

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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.
```

Γ'.3.17 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

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

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

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

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

Γ'.3.18 W3C C14N σουίτα δοκιμής

Η σουίτα δοκιμής C14N 2.0 στο πακέτο `test` (`Lib/test/xmltestdata/c14n-20/`) ανακτήθηκε από τον ιστότοπο του W3C https://www.w3.org/TR/xml-c14n2-testcases/ και διανέμεται με την άδεια 3 ρήτρων BSD:

```
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:

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
* 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.
```

ΠΑΡΑΡΤΗΜΑ Δʹ

Copyright

Ανατρέξτε στο *Ιστορία και Άδεια* για πλήρης πληροφόρηση σχετικά με την άδεια χρήσης και τις εξουσιοδοτήσεις.

Ευρετήριο

μη-αλφαβιτικά

Α

Γ

Δ

Σ

Τ

Χ