

---

# The Python Language Reference

*Δημοσίευση 3.14.0rc2*

**Guido van Rossum and the Python development team**

**Σεπτεμβρίου 01, 2025**

**Python Software Foundation  
Email: [docs@python.org](mailto:docs@python.org)**



<b>1</b>	<b>Εισαγωγή</b>	<b>3</b>
1.1	Εναλλακτικές Υλοποιήσεις	3
1.2	Σημειογραφία	4
1.2.1	Λεξικοί και Συντακτικοί ορισμοί	5
<b>2</b>	<b>Lexical analysis</b>	<b>7</b>
2.1	Line structure	7
2.1.1	Logical lines	7
2.1.2	Physical lines	7
2.1.3	Comments	8
2.1.4	Encoding declarations	8
2.1.5	Explicit line joining	8
2.1.6	Implicit line joining	8
2.1.7	Blank lines	9
2.1.8	Indentation	9
2.1.9	Whitespace between tokens	10
2.1.10	End marker	10
2.2	Other tokens	10
2.3	Names (identifiers and keywords)	10
2.3.1	Keywords	11
2.3.2	Soft Keywords	11
2.3.3	Reserved classes of identifiers	11
2.4	Literals	12
2.5	String and Bytes literals	12
2.5.1	Triple-quoted strings	12
2.5.2	String prefixes	13
2.5.3	Formal grammar	13
2.5.4	Escape sequences	14
2.5.5	Bytes literals	16
2.5.6	Raw string literals	16
2.5.7	f-strings	16
2.5.8	t-strings	18
2.6	Numeric literals	18
2.6.1	Integer literals	19
2.6.2	Floating-point literals	20
2.6.3	Imaginary literals	20
2.7	Operators	21
2.8	Delimiters	21
<b>3</b>	<b>Data model</b>	<b>23</b>
3.1	Objects, values and types	23

3.2	The standard type hierarchy	24
3.2.1	None	24
3.2.2	NotImplemented	24
3.2.3	Ellipsis	24
3.2.4	numbers.Number	25
3.2.5	Sequences	26
3.2.6	Set types	27
3.2.7	Mappings	27
3.2.8	Callable types	27
3.2.9	Modules	30
3.2.10	Custom classes	33
3.2.11	Class instances	35
3.2.12	I/O objects (also known as file objects)	35
3.2.13	Internal types	35
3.3	Special method names	41
3.3.1	Basic customization	42
3.3.2	Customizing attribute access	45
3.3.3	Customizing class creation	49
3.3.4	Customizing instance and subclass checks	53
3.3.5	Emulating generic types	53
3.3.6	Emulating callable objects	55
3.3.7	Emulating container types	55
3.3.8	Emulating numeric types	57
3.3.9	With Statement Context Managers	59
3.3.10	Customizing positional arguments in class pattern matching	60
3.3.11	Emulating buffer types	60
3.3.12	Annotations	61
3.3.13	Special method lookup	62
3.4	Coroutines	63
3.4.1	Awaitable Objects	63
3.4.2	Coroutine Objects	63
3.4.3	Asynchronous Iterators	64
3.4.4	Asynchronous Context Managers	64
<b>4</b>	<b>Μοντέλο εκτέλεσης</b>	<b>67</b>
4.1	Δομή ενός προγράμματος	67
4.2	Ονομασία και σύνδεση	67
4.2.1	Σύνδεση ονομάτων	67
4.2.2	Επίλυση ονομάτων	68
4.2.3	Σημειογραφία πεδία	69
4.2.4	Καθυστερημένη εκτίμηση	70
4.2.5	Ενσωματωμένες συναρτήσεις και περιορισμένη εκτέλεση	71
4.2.6	Αλληλεπίδραση με δυναμικές λειτουργίες	71
4.3	Εξαιρέσεις	71
<b>5</b>	<b>The import system</b>	<b>73</b>
5.1	importlib	73
5.2	Packages	74
5.2.1	Regular packages	74
5.2.2	Namespace packages	74
5.3	Searching	75
5.3.1	The module cache	75
5.3.2	Finders and loaders	75
5.3.3	Import hooks	75
5.3.4	The meta path	76
5.4	Loading	76
5.4.1	Loaders	77
5.4.2	Submodules	78

5.4.3	Module specs	79
5.4.4	__path__ attributes on modules	79
5.4.5	Module reprs	79
5.4.6	Cached bytecode invalidation	79
5.5	The Path Based Finder	80
5.5.1	Path entry finders	80
5.5.2	Path entry finder protocol	81
5.6	Replacing the standard import system	82
5.7	Package Relative Imports	82
5.8	Special considerations for __main__	82
5.8.1	__main__.spec	83
5.9	References	83
<b>6</b>	<b>Expressions</b>	<b>85</b>
6.1	Arithmetic conversions	85
6.2	Atoms	85
6.2.1	Identifiers (Names)	85
6.2.2	Literals	86
6.2.3	Parenthesized forms	87
6.2.4	Displays for lists, sets and dictionaries	87
6.2.5	List displays	88
6.2.6	Set displays	88
6.2.7	Dictionary displays	88
6.2.8	Generator expressions	89
6.2.9	Yield expressions	89
6.3	Primaries	93
6.3.1	Attribute references	94
6.3.2	Subscriptions	94
6.3.3	Slicings	94
6.3.4	Calls	95
6.4	Await expression	97
6.5	The power operator	97
6.6	Unary arithmetic and bitwise operations	97
6.7	Binary arithmetic operations	97
6.8	Shifting operations	99
6.9	Binary bitwise operations	99
6.10	Comparisons	99
6.10.1	Value comparisons	100
6.10.2	Membership test operations	102
6.10.3	Identity comparisons	102
6.11	Boolean operations	102
6.12	Assignment expressions	102
6.13	Conditional expressions	103
6.14	Lambdas	103
6.15	Expression lists	103
6.16	Evaluation order	104
6.17	Operator precedence	104
<b>7</b>	<b>Simple statements</b>	<b>105</b>
7.1	Expression statements	105
7.2	Assignment statements	106
7.2.1	Augmented assignment statements	107
7.2.2	Annotated assignment statements	108
7.3	The assert statement	109
7.4	The pass statement	109
7.5	The del statement	109
7.6	The return statement	109
7.7	The yield statement	110

7.8	The raise statement . . . . .	110
7.9	The break statement . . . . .	112
7.10	The continue statement . . . . .	112
7.11	The import statement . . . . .	112
7.11.1	Future statements . . . . .	114
7.12	The global statement . . . . .	115
7.13	The nonlocal statement . . . . .	115
7.14	The type statement . . . . .	115
<b>8</b>	<b>Compound statements</b>	<b>117</b>
8.1	The if statement . . . . .	118
8.2	The while statement . . . . .	118
8.3	The for statement . . . . .	118
8.4	The try statement . . . . .	119
8.4.1	except clause . . . . .	119
8.4.2	except * clause . . . . .	120
8.4.3	else clause . . . . .	121
8.4.4	finally clause . . . . .	121
8.5	The with statement . . . . .	122
8.6	The match statement . . . . .	123
8.6.1	Overview . . . . .	124
8.6.2	Guards . . . . .	125
8.6.3	Irrefutable Case Blocks . . . . .	125
8.6.4	Patterns . . . . .	125
8.7	Function definitions . . . . .	131
8.8	Class definitions . . . . .	133
8.9	Coroutines . . . . .	134
8.9.1	Coroutine function definition . . . . .	134
8.9.2	The async for statement . . . . .	135
8.9.3	The async with statement . . . . .	135
8.10	Type parameter lists . . . . .	136
8.10.1	Generic functions . . . . .	137
8.10.2	Generic classes . . . . .	138
8.10.3	Generic type aliases . . . . .	139
8.11	Annotations . . . . .	139
<b>9</b>	<b>Top-level components</b>	<b>141</b>
9.1	Complete Python programs . . . . .	141
9.2	File input . . . . .	141
9.3	Interactive input . . . . .	142
9.4	Expression input . . . . .	142
<b>10</b>	<b>Πλήρης προδιαγραφή γραμματικής</b>	<b>143</b>
<b>A'</b>	<b>Γλωσσάρι</b>	<b>161</b>
<b>B'</b>	<b>Σχετικά με την τεκμηρίωση</b>	<b>183</b>
B'.1	Συντελεστές στη τεκμηρίωση της Python . . . . .	183
<b>Γ'</b>	<b>Ιστορία και Άδεια</b>	<b>185</b>
Γ'.1	Η ιστορία του λογισμικού . . . . .	185
Γ'.2	Όροι και προϋποθέσεις για την πρόσβαση ή την χρήση της Python με άλλους τρόπους . . . . .	186
Γ'.2.1	PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 . . . . .	186
Γ'.2.2	ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ BEOPEN.COM ΓΙΑ PYTHON 2.0 . . . . .	187
Γ'.2.3	ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ CNRI ΓΙΑ PYTHON 1.6.1 . . . . .	188
Γ'.2.4	ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ CWI ΓΙΑ PYTHON 0.9.0 ΕΩΣ 1.2 . . . . .	190
Γ'.2.5	ZERO-CLAUSE BSD ΑΔΕΙΑ ΓΙΑ ΤΟΝ ΚΩΔΙΚΑ ΣΤΗΝ ΤΕΚΜΗΡΙΩΣΗ ΤΗΣ PYTHON . . . . .	191
Γ'.3	Άδειες και Ευχαριστίες για Ενσωματωμένο Λογισμικό . . . . .	191
Γ'.3.1	Mersenne Twister . . . . .	191

Γ'.3.2	Sockets . . . . .	192
Γ'.3.3	Ασύγχρονες socket υπηρεσίες . . . . .	193
Γ'.3.4	Διαχείριση Cookie . . . . .	193
Γ'.3.5	Ανίχνευση εκτέλεσης . . . . .	193
Γ'.3.6	Συναρτήσεις UUencode και UUdecode . . . . .	194
Γ'.3.7	Κλήσεις Απομακρυσμένης Διαδικασίας XML . . . . .	195
Γ'.3.8	test_epoll . . . . .	195
Γ'.3.9	Επιλογή kqueue . . . . .	196
Γ'.3.10	SipHash24 . . . . .	196
Γ'.3.11	strtod και dtoa . . . . .	197
Γ'.3.12	OpenSSL . . . . .	197
Γ'.3.13	expat . . . . .	200
Γ'.3.14	libffi . . . . .	201
Γ'.3.15	zlib . . . . .	201
Γ'.3.16	cfuhash . . . . .	202
Γ'.3.17	libmpdec . . . . .	203
Γ'.3.18	W3C C14N σουίτα δοκιμής . . . . .	203
Γ'.3.19	mimalloc . . . . .	204
Γ'.3.20	asyncio . . . . .	204
Γ'.3.21	Καθολικές Απεριόριστες Ακολουθίες (ΚΑΑ) . . . . .	205
Γ'.3.22	Δεσμεύσεις Zstandard . . . . .	205
<b>Δ' Copyright</b>		<b>207</b>
<b>Ευρετήριο</b>		<b>209</b>





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.



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

Η υλοποίηση της Python στην Java. Αυτή η υλοποίηση μπορεί να χρησιμοποιηθεί ως γλώσσα δέσμης ενεργειών για εφαρμογές στην Java, ή μπορεί να χρησιμοποιηθεί για να δημιουργήσει εφαρμογές με

τη χρήση των βιβλιοθηκών των κλάσεων της Java. Συχνά επίσης χρησιμοποιείται για να δημιουργήσει τεστ για τις βιβλιοθήκες της Java. Περισσότερες πληροφορίες μπορείτε να βρείτε στην [ιστοσελίδα της Jython](#).

### 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

Μια εναλλακτική Python για το .NET. Σε αντίθεση με το Python.NET, αυτή είναι μία ολοκληρωμένη υλοποίηση της Python που παράγει IL, και κάνει μεταγλώττιση του κώδικα της Python απευθείας στη γλώσσα assembly του .NET. Δημιουργήθηκε από τον Jim Hugunin, τον πρωτότυπο δημιουργό της Jython. Για περισσότερες πληροφορίες δείτε την [ιστοσελίδα της IronPython](#).

### PyPy

Μια υλοποίηση της Python γραμμένη εξ ολοκλήρου σε Python. Υποστηρίζει αρκετές προηγμένες λειτουργίες που δεν υπάρχουν σε άλλες υλοποιήσεις όπως υποστήριξη για stackless και τον μεταγλωττιστή Just in Time. Ένας από τους στόχους του πρότζεκτ είναι να ενθαρρύνει τον πειραματισμό με την ίδια την γλώσσα κάνοντας πιο εύκολη την τροποποίηση του διερμηνέα (αφού είναι γραμμένος στην Python). Περισσότερες πληροφορίες είναι διαθέσιμες στην [αρχική σελίδα του PyPy πρότζεκτ](#).

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

Οι περιγραφές στην λεξιλογική ανάλυση και σύνταξη χρησιμοποιούν μια γραμματική σημειογραφία που είναι μείγμα των EBNF και PEG. Για παράδειγμα:

```
name:    letter (letter | digit | "_") *
letter:  "a"..."z" | "A"..."Z"
digit:   "0"..."9"
```

Σε αυτό το παράδειγμα, η πρώτη γραμμή αναφέρει ότι ένα `name` είναι ένα `letter` ακολουθούμενο από μια ακολουθία μηδενός ή περισσότερων γραμμάτων, ψηφία, και κάτω παύλων. Ένα `letter` με τη σειρά του είναι οποιοσδήποτε από τους μεμονωμένους χαρακτήρες 'a' έως 'z' και A έως Z ``. Ένα ``digit είναι ένας μεμονωμένος χαρακτήρας από 0 έως 9.

Κάθε κανόνας ξεκινά με ένα όνομα (το οποίο προσδιορίζει τον κανόνα που ορίζεται) ακολουθούμενο από άνω κάτω τελεία, `:`. Ο ορισμός στα δεξιά της άνω και κάτω τελείας χρησιμοποιεί τα ακόλουθα στοιχεία σύνταξης:

- `name`: Ένα όνομα αναφέρεται σε έναν άλλο κανόνα, Όπου είναι δυνατόν, είναι ένας σύνδεσμος προς τον ορισμό του κανόνα.
  - `TOKEN`: Ένα κεφαλαίο όνομα αναφέρεται σε ένα *token*. Για τους σκοπούς των γραμματικών ορισμών, τα `tokens` είναι το ίδιο με τους κανόνες.
- `"text"`, `'text'`: Το κείμενο σε μονά ή διπλά εισαγωγικά πρέπει να ταιριάζει κυριολεκτικά (χωρίς τα εισαγωγικά). Ο τύπος του εισαγωγικού επιλέγεται ανάλογα με τη σημασία του `text`:
  - `'if'`: Ένα όνομα σε μονά εισαγωγικά υποδηλώνει μια *keyword*.
  - `"case"`: Ένα όνομα σε διπλά εισαγωγικά υποδηλώνει ένα *soft-keyword*.
  - `'@'`: Ένα σύμβολο που δεν περιέχει γράμμα σε μονά εισαγωγικά υποδηλώνει ένα `OP token`, δηλαδή, ένα *delimiter* ή *operator*.
- `e1 e2`: Τα στοιχεία που χωρίζονται μόνο με κενό υποδηλώνουν μια ακολουθία. Εδώ, το `e1` πρέπει να ακολουθείται από το `e2`.

- `e1 | e2`: Χρησιμοποιείται μια κάθετη γραμμή για τον διαχωρισμό των εναλλακτικών λύσεων. Υποδηλώνει την «διατεταγμένη επιλογή» του PEG: εάν το `e1` ταιριάζει, το `e2` δεν λαμβάνεται υπόψη. Στις παραδοσιακές γραμματικές του PEG, αυτό γράφεται ως κάθετος, `/`, αντί για κάθετη γραμμή. Δείτε το [PEP 617](#) για περισσότερες πληροφορίες και λεπτομέρειες.
- `e*`: Ένας αστερίσκος σημαίνει μηδέν ή περισσότερες επαναλήψεις του προηγούμενου στοιχείου.
- `e+`: Ομοίως, ένα συν σημαίνει μία ή περισσότερες επαναλήψεις.
- `[e]`: Μια φράση που περικλείεται σε αγκύλες σημαίνει μηδέν ή μία εμφάνιση. Με άλλα λόγια, η φράση που περικλείεται είναι προαιρετική.
- `e?`: Ένα ερωτηματικό έχει ακριβώς την ίδια σημασία με τις αγκύλες: το προηγούμενο στοιχείο είναι προαιρετικό.
- `(e)`: Οι παρενθέσεις χρησιμοποιούνται για ομαδοποίηση.

The following notation is only used in *lexical definitions*.

- `"a" . . . "z"`: Two literal characters separated by three dots mean a choice of any single character in the given (inclusive) range of ASCII characters.
- `< . . . >`: A phrase between angular brackets gives an informal description of the matched symbol (for example, `<any ASCII character except "\">`), or an abbreviation that is defined in nearby text (for example, `<Lu>`).

Some definitions also use *lookaheads*, which indicate that an element must (or must not) match at a given position, but without consuming any input:

- `&e`: a positive lookahead (that is, `e` is required to match)
- `!e`: a negative lookahead (that is, `e` is required *not* to match)

Οι μοναδιαίοι τελεστές (`*`, `+`, `?`) συνδέονται όσο το δυνατόν πιο σφιχτά• η κάθετη γραμμή (`|`) συνδέεται πιο χαλαρά.

Το κενό έχει νόημα μόνο για τον διαχωρισμό των διακριτικών.

Οι κανόνες συνήθως περιέχονται σε μία μόνο γραμμή, αλλά οι κανόνες που είναι πολύ μεγάλοι μπορούν να αναδιπλωθούν.

```
literal: stringliteral | bytesliteral
        | integer | floatnumber | imagnumber
```

Εναλλακτικά, οι κανόνες μπορούν να μορφοποιηθούν με την πρώτη γραμμή να τελειών στην άνω και κάτω τελεία και κάθε εναλλακτική να ξεκινά με μια κάθετη γραμμή σε μια νέα γραμμή. Για παράδειγμα:

```
literal:
    | stringliteral
    | bytesliteral
    | integer
    | floatnumber
    | imagnumber
```

Αυτό δεν σημαίνει ότι υπάρχει μια κενή πρώτη εναλλακτική λύση.

### 1.2.1 Λεξικοί και Συντακτικοί ορισμοί

Υπάρχει κάποια διαφορά μεταξύ της *λεξικής* και της *συντακτικής* ανάλυσης: ο *lexical analyzer* λειτουργεί στους μεμονωμένους χαρακτήρες της πηγής εισόδου, ενώ ο *αναλυτής* (συντακτικός αναλυτής) λειτουργεί στη ροή των *tokens* που δημιουργούνται από τη λεξική ανάλυση. Ωστόσο, σε ορισμένες περιπτώσεις το ακριβές όριο μεταξύ των δύο φάσεων είναι μια λεπτομέρεια υλοποίησης της CPython.

Η πρακτική διαφορά μεταξύ των δύο είναι ότι στους *λεξιλογικούς* ορισμούς, όλα τα κενά είναι σημαντικά. Ο λεξικός αναλυτής *discards* όλα τα κενά που δεν μετατρέπονται σε διακριτικά όπως `token.INDENT` ή `NEWLINE`. Οι *συντακτικοί* ορισμοί χρησιμοποιούν στη συνέχεια αυτά τα διακριτικά, αντί για τους χαρακτήρες προέλευσης.

Αυτή η τεκμηρίωση χρησιμοποιεί την ίδια γραμματική BNF και για τα δύο στυλ ορισμών. Όλες οι χρήσεις του BNF στο επόμενο κεφάλαιο (*Lexical analysis*) είναι λεξιλογικοί ορισμοί• οι χρήσεις στα επόμενα κεφάλαια είναι συντακτικοί ορισμοί.

A Python program is read by a *parser*. Input to the parser is a stream of *tokens*, generated by the *lexical analyzer* (also known as the *tokenizer*). 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 one the following end-of-line sequences:

- the Unix form using ASCII LF (linefeed),
- the Windows form using the ASCII sequence CR LF (return followed by linefeed),
- the “Classic Mac OS” form using the ASCII CR (return) character.

Regardless of platform, each of these sequences is replaced by a single ASCII LF (linefeed) character. (This is done even inside *string literals*.) Each line can use any of the sequences; they do not need to be consistent within a file.

The end of input also serves as an implicit terminator for the final physical line.

Formally:

```
newline: <ASCII LF> | <ASCII CR> <ASCII LF> | <ASCII CR>
```

### 2.1.3 Comments

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

### 2.1.4 Encoding declarations

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

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

which is recognized also by GNU Emacs, and

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

which is recognized by Bram Moolenaar's VIM.

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

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

All lexical analysis, including string literals, comments and identifiers, works on Unicode text decoded using the source encoding. Any Unicode code point, except the NUL control character, can appear in Python source.

```
source_character: <any Unicode code point, except NUL>
```

### 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 (that is, 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+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+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. For example, `ab` is one token, but `a b` is two tokens. However, `+a` and `+ a` both produce two tokens, `+` and `a`, as `+a` is not a valid token.

### 2.1.10 End marker

At the end of non-interactive input, the lexical analyzer generates an `ENDMARKER` token.

## 2.2 Other tokens

Besides `NEWLINE`, `INDENT` and `DEDENT`, the following categories of tokens exist: *identifiers* and *keywords* (`NAME`), *literals* (such as `NUMBER` and `STRING`), and other symbols (*operators* and *delimiters*, `OP`). Whitespace characters (other than logical 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 Names (identifiers and keywords)

`NAME` tokens represent *identifiers*, *keywords*, and *soft keywords*.

Within the ASCII range (U+0001..U+007F), the valid characters for names include the uppercase and lowercase letters (A–Z and a–z), the underscore `_` and, except for the first character, the digits 0 through 9.

Names must contain at least one character, but have no upper length limit. Case is significant.

Besides A–Z, a–z, `_` and 0–9, names can also use «letter-like» and «number-like» characters from outside the ASCII range, as detailed below.

All identifiers are converted into the [normalization form](#) NFKC while parsing; comparison of identifiers is based on NFKC.

Formally, the first character of a normalized identifier must belong to the set `id_start`, which is the union of:

- Unicode category `<Lu>` - uppercase letters (includes A to Z)
- Unicode category `<Ll>` - lowercase letters (includes a to z)
- Unicode category `<Lt>` - titlecase letters
- Unicode category `<Lm>` - modifier letters
- Unicode category `<Lo>` - other letters
- Unicode category `<Nl>` - letter numbers
- `{"_"}` - the underscore
- `<Other_ID_Start>` - an explicit set of characters in [PropList.txt](#) to support backwards compatibility

The remaining characters must belong to the set `id_continue`, which is the union of:

- all characters in `id_start`
- Unicode category `<Nd>` - decimal numbers (includes 0 to 9)
- Unicode category `<Pc>` - connector punctuations
- Unicode category `<Mn>` - nonspacing marks
- Unicode category `<Mc>` - spacing combining marks

- `<Other_ID_Continue>` - another explicit set of characters in [PropList.txt](#) to support backwards compatibility

Unicode categories use the version of the Unicode Character Database as included in the `unicodedata` module.

These sets are based on the Unicode standard annex [UAX-31](#). See also [PEP 3131](#) for further details.

Even more formally, names are described by the following lexical definitions:

```
NAME:           xid_start xid_continue*
id_start:       <Lu> | <Ll> | <Lt> | <Lm> | <Lo> | <Nl> | "_" | <Other_ID_Start>
id_continue:    id_start | <Nd> | <Pc> | <Mn> | <Mc> | <Other_ID_Continue>
xid_start:      <all characters in id_start whose NFKC normalization is
                 in (id_start xid_continue*) ">
xid_continue:   <all characters in id_continue whose NFKC normalization is
                 in (id_continue*) ">
identifier:     <NAME, except keywords>
```

A non-normative listing of all valid identifier characters as defined by Unicode is available in the [DerivedCoreProperties.txt](#) file in the Unicode Character Database.

### 2.3.1 Keywords

The following names 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 Soft Keywords

Added in version 3.10.

Some names are only reserved under specific contexts. These are known as *soft keywords*:

- `match`, `case`, and `_`, when used in the `match` statement.
- `type`, when used in the `type` statement.

These syntactically act as keywords in their specific contexts, but this distinction is done at the parser level, not when tokenizing.

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

Αλλάξε στην έκδοση 3.12: `type` is now a soft keyword.

### 2.3.3 Reserved classes of identifiers

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

— \*

Not imported by `from module import *`.

—

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

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

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

#### Σημείωση

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

It is also commonly used for unused variables.

#### `__*`

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

#### `__*`

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

## 2.4 Literals

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

In terms of lexical analysis, Python has *string*, *bytes* and *numeric* literals.

Other «literals» are lexically denoted using *keywords* (`None`, `True`, `False`) and the special *ellipsis token* (`...`).

## 2.5 String and Bytes literals

String literals are text enclosed in single quotes (`'`) or double quotes (`"`). For example:

```
"spam"  
'eggs'
```

The quote used to start the literal also terminates it, so a string literal can only contain the other quote (except with escape sequences, see below). For example:

```
'Say "Hello", please.'  
"Don't do that!"
```

Except for this limitation, the choice of quote character (`'` or `"`) does not affect how the literal is parsed.

Inside a string literal, the backslash (`\`) character introduces an *escape sequence*, which has special meaning depending on the character after the backslash. For example, `\"` denotes the double quote character, and does *not* end the string:

```
>>> print("Say \"Hello\" to everyone!")  
Say "Hello" to everyone!
```

See [escape sequences](#) below for a full list of such sequences, and more details.

### 2.5.1 Triple-quoted strings

Strings can also be enclosed in matching groups of three single or double quotes. These are generally referred to as *triple-quoted strings*:

```
"""This is a triple-quoted string."""
```

In triple-quoted literals, unescaped quotes are allowed (and are retained), except that three unescaped quotes in a row terminate the literal, if they are of the same kind ( ' or ") used at the start:

```
"""This string has "quotes" inside."""
```

Unescaped newlines are also allowed and retained:

```
'''This triple-quoted string
continues on the next line.'''
```

## 2.5.2 String prefixes

String literals can have an optional *prefix* that influences how the content of the literal is parsed, for example:

```
b"data"
f'{result=}'
```

The allowed prefixes are:

- `b`: *Bytes literal*
- `r`: *Raw string*
- `f`: *Formatted string literal* («f-string»)
- `t`: *Template string literal* («t-string»)
- `u`: No effect (allowed for backwards compatibility)

See the linked sections for details on each type.

Prefixes are case-insensitive (for example, “B” works the same as “b”). The “r” prefix can be combined with “f”, “t” or “b”, so “fr”, “rf”, “tr”, “rt”, “br”, and “rb” are also valid prefixes.

Added in version 3.3: The `'rb'` prefix of raw bytes literals has been added as a synonym of `'br'`.

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

## 2.5.3 Formal grammar

String literals, except «f-strings» and «t-strings», are described by the following lexical definitions.

These definitions use *negative lookaheads* (!) to indicate that an ending quote ends the literal.

```
STRING:          [stringprefix] (stringcontent)
stringprefix:    <("r" | "u" | "b" | "br" | "rb"), case-insensitive>
stringcontent:
    | " ( !" stringitem) * "
    | ' ( !' stringitem) * '
    | " " ( !" longstringitem) * " "
    | ' ' ( !' longstringitem) * ' '
stringitem:      stringchar | stringescapeseq
stringchar:      <any source_character, except backslash and newline>
longstringitem:  stringitem | newline
stringescapeseq: " \ " <any source_character>
```

Note that as in all lexical definitions, whitespace is significant. In particular, the prefix (if any) must be immediately followed by the starting quote.

## 2.5.4 Escape sequences

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
\<newline>	<i>Ignored end of line</i>
\\	<i>Backslash</i>
\'	<i>Single quote</i>
\"	<i>Double quote</i>
\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	<i>Octal character</i>
\xhh	<i>Hexadecimal character</i>
\N{name}	<i>Named Unicode character</i>
\uxxxx	<i>Hexadecimal Unicode character</i>
\Uxxxxxxxx	<i>Hexadecimal Unicode character</i>

### Ignored end of line

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

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

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

### Escaped characters

To include a backslash in a non-*raw* Python string literal, it must be doubled. The `\\` escape sequence denotes a single backslash character:

```
>>> print('C:\\Program Files')
C:\Program Files
```

Similarly, the `\'` and `\"` sequences denote the single and double quote character, respectively:

```
>>> print('\ ' and '\"')
' and "
```

### Octal character

The sequence `\ooo` denotes a *character* with the octal (base 8) value *ooo*:

```
>>> '\120'
'P'
```

Up to three octal digits (0 through 7) are accepted.

In a bytes literal, *character* means a *byte* with the given value. In a string literal, it means a Unicode character with the given value.

Αλλάξε στην έκδοση 3.11: Octal escapes with value larger than `0o377` (255) produce a `DeprecationWarning`.

Αλλάξε στην έκδοση 3.12: Octal escapes with value larger than 0o377 (255) produce a `SyntaxWarning`. In a future Python version they will raise a `SyntaxError`.

### Hexadecimal character

The sequence `\xhh` denotes a *character* with the hex (base 16) value *hh*:

```
>>> '\x50'
'P'
```

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

In a bytes literal, *character* means a *byte* with the given value. In a string literal, it means a Unicode character with the given value.

### Named Unicode character

The sequence `\N{name}` denotes a Unicode character with the given *name*:

```
>>> '\N{LATIN CAPITAL LETTER P}'
'P'
>>> '\N{SNAKE}'
'␣'
```

This sequence cannot appear in *bytes literals*.

Αλλάξε στην έκδοση 3.3: Support for *name aliases* has been added.

### Hexadecimal Unicode characters

These sequences `\uxxxx` and `\Uxxxxxxxx` denote the Unicode character with the given hex (base 16) value. Exactly four digits are required for `\u`; exactly eight digits are required for `\U`. The latter can encode any Unicode character.

```
>>> '\u1234'
'␣'
>>> '\U0001f40d'
'🍝'
```

These sequences cannot appear in *bytes literals*.

### Unrecognized escape sequences

Unlike in Standard C, all unrecognized escape sequences are left in the string unchanged, that is, *the backslash is left in the result*:

```
>>> print('\q')
\q
>>> list('\q')
['\\', 'q']
```

Note that for bytes literals, the escape sequences only recognized in string literals (`\N...`, `\u...`, `\U...`) fall into the category of unrecognized escapes.

Αλλάξε στην έκδοση 3.6: Unrecognized escape sequences produce a `DeprecationWarning`.

Αλλάξε στην έκδοση 3.12: Unrecognized escape sequences produce a `SyntaxWarning`. In a future Python version they will raise a `SyntaxError`.

## 2.5.5 Bytes literals

*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 escape sequences (typically *Hexadecimal character* or *Octal character*):

```
>>> b'\x89PNG\r\n\x1a\n'
b'\x89PNG\r\n\x1a\n'
>>> list(b'\x89PNG\r\n\x1a\n')
[137, 80, 78, 71, 13, 10, 26, 10]
```

Similarly, a zero byte must be expressed using an escape sequence (typically `\0` or `\x00`).

## 2.5.6 Raw string literals

Both string and bytes literals may optionally be prefixed with a letter “r” or “R”; such constructs are called *raw string literals* and *raw bytes literals* respectively and treat backslashes as literal characters. As a result, in raw string literals, *escape sequences* are not treated specially:

```
>>> r'\d{4}-\d{2}-\d{2}'
'\\d{4}-\\d{2}-\\d{2}'
```

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.

## 2.5.7 f-strings

Added in version 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 | replacement_field)*
literal_char:  <any code point except "{", "}" or NULL>
```

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

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



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

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.

Άλλαξε στην έκδοση 3.12: Prior to Python 3.12, comments were not allowed inside f-string replacement fields.

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

Added in version 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" '
```

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

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

Αλλάξε στην έκδοση 3.12: Prior to Python 3.12, reuse of the same quoting type of the outer f-string inside a replacement field was not possible.

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

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

Αλλάξε στην έκδοση 3.12: Prior to Python 3.12, backslashes were not permitted inside an f-string replacement field.

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

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

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

## 2.5.8 t-strings

Added in version 3.14.

A *template string literal* or *t-string* is a string literal that is prefixed with “t” or “T”. These strings follow the same syntax and evaluation rules as *formatted string literals*, with the following differences:

- Rather than evaluating to a `str` object, template string literals evaluate to a `string.Template` object.
- The `format()` protocol is not used. Instead, the format specifier and conversions (if any) are passed to a new `Interpolation` object that is created for each evaluated expression. It is up to code that processes the resulting `Template` object to decide how to handle format specifiers and conversions.
- Format specifiers containing nested replacement fields are evaluated eagerly, prior to being passed to the `Interpolation` object. For instance, an interpolation of the form `{amount:.{precision}f}` will evaluate the inner expression `{precision}` to determine the value of the `format_spec` attribute. If `precision` were to be 2, the resulting format specifier would be `'.2f'`.
- When the equals sign `'='` is provided in an interpolation expression, the text of the expression is appended to the literal string that precedes the relevant interpolation. This includes the equals sign and any surrounding whitespace. The `Interpolation` instance for the expression will be created as normal, except that conversion will be set to `"r"` (`repr()`) by default. If an explicit conversion or format specifier are provided, this will override the default behaviour.

## 2.6 Numeric literals

`NUMBER` tokens represent numeric literals, of which there are three types: integers, floating-point numbers, and imaginary numbers.

```
NUMBER: integer | floatnumber | imagnumber
```

The numeric value of a numeric literal is the same as if it were passed as a string to the `int`, `float` or `complex` class constructor, respectively. Note that not all valid inputs for those constructors are also valid literals.

Numeric literals do not include a sign; a phrase like `-1` is actually an expression composed of the unary operator “`-`” and the literal `1`.

### 2.6.1 Integer literals

Integer literals denote whole numbers. For example:

```
7
3
2147483647
```

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

```
7922816251426433759354395033679228162514264337593543950336
```

Underscores can be used to group digits for enhanced readability, and are ignored for determining the numeric value of the literal. For example, the following literals are equivalent:

```
100_000_000_000
100000000000
1_00_00_00_00_000
```

Underscores can only occur between digits. For example, `_123`, `321_`, and `123__321` are *not* valid literals.

Integers can be specified in binary (base 2), octal (base 8), or hexadecimal (base 16) using the prefixes `0b`, `0o` and `0x`, respectively. Hexadecimal digits 10 through 15 are represented by letters A-F, case-insensitive. For example:

```
0b100110111
0b_1110_0101
0o177
0o377
0xdeadbeef
0xDead_Beef
```

An underscore can follow the base specifier. For example, `0x_1f` is a valid literal, but `0_x1f` and `0x__1f` are not.

Leading zeros in a non-zero decimal number are not allowed. For example, `0123` is not a valid literal. This is for disambiguation with C-style octal literals, which Python used before version 3.0.

Formally, integer literals are described by the following lexical definitions:

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

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

## 2.6.2 Floating-point literals

Floating-point (float) literals, such as `3.14` or `1.5`, denote *approximations of real numbers*.

They consist of *integer* and *fraction* parts, each composed of decimal digits. The parts are separated by a decimal point, `.`:

```
2.71828
4.0
```

Unlike in integer literals, leading zeros are allowed in the numeric parts. For example, `077.010` is legal, and denotes the same number as `77.10`.

As in integer literals, single underscores may occur between digits to help readability:

```
96_485.332_123
3.14_15_93
```

Either of these parts, but not both, can be empty. For example:

```
10. # (equivalent to 10.0)
.001 # (equivalent to 0.001)
```

Optionally, the integer and fraction may be followed by an *exponent*: the letter `e` or `E`, followed by an optional sign, `+` or `-`, and a number in the same format as the integer and fraction parts. The `e` or `E` represents «times ten raised to the power of»:

```
1.0e3 # (represents 1.0×103, or 1000.0)
1.166e-5 # (represents 1.166×10-5, or 0.00001166)
6.02214076e+23 # (represents 6.02214076×1023, or 602214076000000000000000.
→)
```

In floats with only integer and exponent parts, the decimal point may be omitted:

```
1e3 # (equivalent to 1.e3 and 1.0e3)
0e0 # (equivalent to 0.)
```

Formally, floating-point literals are described by the following lexical definitions:

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

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

## 2.6.3 Imaginary literals

Python has complex number objects, but no complex literals. Instead, *imaginary literals* denote complex numbers with a zero real part.

For example, in math, the complex number  $3+4.2i$  is written as the real number 3 added to the imaginary number  $4.2i$ . Python uses a similar syntax, except the imaginary unit is written as `j` rather than `i`:

```
3+4.2j
```

This is an expression composed of the *integer literal* `3`, the *operator* `“+”`, and the *imaginary literal* `4.2j`. Since these are three separate tokens, whitespace is allowed between them:



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:

\$        ?        `

### 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.

**Λεπτομέρεια υλοποίησης CPython:** 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.<sup>1</sup>

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.

**Λεπτομέρεια υλοποίησης CPython:** 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).

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.

---

<sup>1</sup> 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.

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. For example, after `a = 1; b = 1`, `a` and `b` may or may not refer to the same object with the value one, depending on the implementation. This is because `int` is an immutable type, so the reference to `1` can be reused. This behaviour depends on the implementation used, so should not be relied upon, but is something to be aware of when making use of object identity tests. However, after `c = []; d = []`, `c` and `d` are guaranteed to refer to two different, unique, newly created empty lists. (Note that `e = f = []` assigns the *same* object to both `e` and `f`.)

## 3.2 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.

### 3.2.1 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.

### 3.2.2 NotImplemented

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

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

Άλλαξε στην έκδοση 3.9: Evaluating `NotImplemented` in a boolean context was deprecated.

Άλλαξε στην έκδοση 3.14: Evaluating `NotImplemented` in a boolean context now raises a `TypeError`. It previously evaluated to `True` and emitted a `DeprecationWarning` since Python 3.9.

### 3.2.3 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.



### 3.2.4 `numbers.Number`

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

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

- They are valid numeric literals which, when passed to their class constructor, produce an object having the value of the original numeric.
- The representation is in base 10, when possible.
- Leading zeros, possibly excepting a single zero before a decimal point, are not shown.
- Trailing zeros, possibly excepting a single zero after a decimal point, are not shown.
- A sign is shown only when the number is negative.

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

#### `numbers.Integral`

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

#### Σημείωση

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

There are two types of integers:

#### **Integers (`int`)**

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

#### **Booleans (`bool`)**

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

#### `numbers.Real (float)`

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

#### `numbers.Complex (complex)`

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

### 3.2.5 Sequences

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

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

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

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  $0 \leq x < 256$ . 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.

#### Σημείωση

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

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.

### 3.2.6 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.

### 3.2.7 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.

Αλλαξε 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.

### 3.2.8 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 read-only attributes

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

## Special writable attributes

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

Attribute	Meaning
<code>function.__doc__</code>	The function's documentation string, or None if unavailable.
<code>function.__name__</code>	The function's name. See also: <code>__name__</code> attributes.
<code>function.__qualname__</code>	The function's <i>qualified name</i> . See also: <code>__qualname__</code> attributes. Added in version 3.3.
<code>function.__module__</code>	The name of the module the function was defined in, or None if unavailable.
<code>function.__defaults__</code>	A tuple containing default <i>parameter</i> values for those parameters that have defaults, or None if no parameters have a default value.
<code>function.__code__</code>	The <i>code object</i> representing the compiled function body.
<code>function.__dict__</code>	The namespace supporting arbitrary function attributes. See also: <code>__dict__</code> attributes.
<code>function.__annotations__</code>	A dictionary containing annotations of <i>parameters</i> . The keys of the dictionary are the parameter names, and 'return' for the return annotation, if provided. See also: <i>object.__annotations__</i> . Άλλαξε στην έκδοση 3.14: Annotations are now <i>lazily evaluated</i> . See <b>PEP 649</b> .
<code>function.__annotate__</code>	The <i>annotate function</i> for this function, or None if the function has no annotations. See <i>object.__annotate__</i> . Added in version 3.14.
<code>function.__kwdefaults__</code>	A dictionary containing defaults for keyword-only <i>parameters</i> .
<code>function.__type_params__</code>	A tuple containing the <i>type parameters</i> of a <i>generic function</i> . Added in version 3.12.

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

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

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

### 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:

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

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

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

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

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

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

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

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

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

## 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`. (The attribute has the same semantics as it does with *other instance methods*.)

## Classes

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

## Class Instances

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

## 3.2.9 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`.

## Import-related attributes on module objects

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

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

### ⚠ Προσοχή

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

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

#### module.\_\_name\_\_

The name used to uniquely identify the module in the import system. For a directly executed module, this will be set to `"__main__"`.

This attribute must be set to the fully qualified name of the module. It is expected to match the value of `module.__spec__.name`.

#### module.\_\_spec\_\_

A record of the module's import-system-related state.

Set to the module `spec` that was used when importing the module. See *Module specs* for more details.

Added in version 3.4.

#### module.\_\_package\_\_

The *package* a module belongs to.

If the module is top-level (that is, not a part of any specific package) then the attribute should be set to `''` (the empty string). Otherwise, it should be set to the name of the module's package (which can be equal to `module.__name__` if the module itself is a package). See **PEP 366** for further details.

This attribute is used instead of `__name__` to calculate explicit relative imports for main modules. It defaults to `None` for modules created dynamically using the `types.ModuleType` constructor; use `importlib.util.module_from_spec()` instead to ensure the attribute is set to a `str`.

It is **strongly** recommended that you use `module.__spec__.parent` instead of `module.__package__`. `__package__` is now only used as a fallback if `__spec__.parent` is not set, and this fallback path is deprecated.

Αλλαξε στην έκδοση 3.4: This attribute now defaults to `None` for modules created dynamically using the `types.ModuleType` constructor. Previously the attribute was optional.

Αλλαξε στην έκδοση 3.6: The value of `__package__` is expected to be the same as `__spec__.parent`. `__package__` is now only used as a fallback during import resolution if `__spec__.parent` is not defined.



Αλλάξε στην έκδοση 3.10: `ImportWarning` is raised if an import resolution falls back to `__package__` instead of `__spec__.parent`.

Αλλάξε στην έκδοση 3.12: Raise `DeprecationWarning` instead of `ImportWarning` when falling back to `__package__` during import resolution.

Deprecated since version 3.13, will be removed in version 3.15: `__package__` will cease to be set or taken into consideration by the import system or standard library.

#### `module.__loader__`

The *loader* object that the import machinery used to load the module.

This attribute is mostly useful for introspection, but can be used for additional loader-specific functionality, for example getting data associated with a loader.

`__loader__` defaults to `None` for modules created dynamically using the `types.ModuleType` constructor; use `importlib.util.module_from_spec()` instead to ensure the attribute is set to a *loader* object.

It is **strongly** recommended that you use `module.__spec__.loader` instead of `module.__loader__`.

Αλλάξε στην έκδοση 3.4: This attribute now defaults to `None` for modules created dynamically using the `types.ModuleType` constructor. Previously the attribute was optional.

Deprecated since version 3.12, will be removed in version 3.16: Setting `__loader__` on a module while failing to set `__spec__.loader` is deprecated. In Python 3.16, `__loader__` will cease to be set or taken into consideration by the import system or the standard library.

#### `module.__path__`

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

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

#### `module.__file__`

#### `module.__cached__`

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

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

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

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

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

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



### Other writable attributes on module objects

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

`module.__doc__`

The module's documentation string, or None if unavailable. See also: `__doc__` attributes.

`module.__annotations__`

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

Άλλαξε στην έκδοση 3.14: Annotations are now *lazily evaluated*. See [PEP 649](#).

`module.__annotate__`

The *annotate function* for this module, or None if the module has no annotations. See also: `__annotate__` attributes.

Added in version 3.14.

### Module dictionaries

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

`module.__dict__`

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

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

### 3.2.10 Custom classes

Custom class types are typically created by class definitions (see section *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 at `python_2.3_mro`.

When a class attribute reference (for class `C`, say) would yield a class method object, it is transformed into an instance method object whose `__self__` attribute is `C`. When it would yield a `staticmethod` object, it is transformed into the object wrapped by the static method object. See section *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

Attribute	Meaning
<code>type.__name__</code>	The class's name. See also: <code>__name__</code> attributes.
<code>type.__qualname__</code>	The class's <i>qualified name</i> . See also: <code>__qualname__</code> attributes.
<code>type.__module__</code>	The name of the module in which the class was defined.
<code>type.__dict__</code>	A mapping proxy providing a read-only view of the class's namespace. See also: <code>__dict__</code> <i>attributes</i> .
<code>type.__bases__</code>	A tuple containing the class's bases. In most cases, for a class defined as <code>class X(A, B, C), X</code> , <code>__bases__</code> will be exactly equal to <code>(A, B, C)</code> .
<code>type.__doc__</code>	The class's documentation string, or <code>None</code> if undefined. Not inherited by subclasses.
<code>type.__annotations__</code>	<p>A dictionary containing <i>variable annotations</i> collected during class body execution. See also: <code>__annotations__</code> <i>attributes</i>.</p> <p>For best practices on working with <code>__annotations__</code>, please see <code>annotationlib</code>. Use <code>annotationlib.get_annotations()</code> instead of accessing this attribute directly.</p> <div data-bbox="821 1164 1385 1527" style="border: 1px solid red; padding: 10px; margin: 10px 0;"> <p><b>⚠ Προειδοποίηση</b></p> <p>Accessing the <code>__annotations__</code> attribute directly on a class object may return annotations for the wrong class, specifically in certain cases where the class, its base class, or a metaclass is defined under <code>from __future__ import annotations</code>. See <a href="#">749</a> for details.</p> <p>This attribute does not exist on certain builtin classes. On user-defined classes without <code>__annotations__</code>, it is an empty dictionary.</p> </div> <p>Αλλάξε στην έκδοση 3.14: Annotations are now <i>lazily evaluated</i>. See <a href="#">PEP 649</a>.</p>
<code>type.__annotate__()</code>	<p>The <i>annotate function</i> for this class, or <code>None</code> if the class has no annotations. See also: <code>__annotate__</code> <i>attributes</i>.</p> <p>Added in version 3.14.</p>
<code>type.__type_params__</code>	<p>A tuple containing the <i>type parameters</i> of a <i>generic class</i>.</p> <p>Added in version 3.12.</p>
<code>type.__static_attributes__</code>	<p>A tuple containing names of attributes of this class which are assigned through <code>self.X</code> from any function in its body.</p> <p>Added in version 3.13.</p>
<code>type.__firstlineno__</code>	<p>The line number of the first line of the class definition, including decorators. Setting the <code>__module__</code> attribute removes the <code>__firstlineno__</code> item from the type's dictionary.</p> <p>Added in version 3.13.</p>
<code>type.__mro__</code>	<p>The tuple of classes that are considered when looking for base classes during method resolution.</p>

## Special methods

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

`type.mro()`

This method can be overridden by a metaclass to customize the method resolution order for its instances. It is called at class instantiation, and its result is stored in `__mro__`.

`type.__subclasses__()`

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

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

### 3.2.11 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

`object.__class__`

The class to which a class instance belongs.

`object.__dict__`

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

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

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

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

### 3.2.13 Internal types

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

## 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

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

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

Future feature declarations (for example, `from __future__ import division`) also use bits in `co_flags` to indicate whether a code object was compiled with a particular feature enabled. See `compiler_flag`.

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

If a code object represents a function and has a docstring, the `CO_HAS_DOCSTRING` bit is set in `co_flags` and the first item in `co_consts` is the docstring of the function.

## Methods on code objects

`codeobject.co_positions()`

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

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

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

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

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

Added in version 3.11.

### Σημείωση

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

`codeobject.co_lines()`

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

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

The items yielded will have the following properties:

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

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

Added in version 3.10.

➡ Δείτε επίσης

#### PEP 626 - Precise line numbers for debugging and other tools.

The PEP that introduced the `co_lines()` method.

`codeobject.replace(**kwargs)`

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

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

Added in version 3.8.

## Frame objects

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

### Special read-only attributes

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

## Special writable attributes

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

## Frame object methods

Frame objects support one method:

`frame.clear()`

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

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

Added in version 3.4.

Αλλάξε στην έκδοση 3.13: Attempting to clear a suspended frame raises `RuntimeError` (as has always been the case for executing frames).

## Traceback objects

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

Αλλάξε στην έκδοση 3.7: Traceback objects can now be explicitly instantiated from Python code.

For implicitly created tracebacks, when the search for an exception handler unwinds the execution stack, at each unwound level a traceback object is inserted in front of the current traceback. When an exception handler is entered, the stack trace is made available to the program. (See section [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:



<code>traceback.tb_frame</code>	Points to the execution <i>frame</i> of the current level. Accessing this attribute raises an auditing event object.__getattr__ with arguments obj and "tb_frame".
<code>traceback.tb_lineno</code>	Gives the line number where the exception occurred
<code>traceback.tb_lasti</code>	Indicates the «precise instruction».

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

`traceback.tb_next`

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

Άλλαξε στην έκδοση 3.7: This attribute is now writable

## Slice objects

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

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

Slice objects support one method:

`slice.indices(self, length)`

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

## Static method objects

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

## Class method objects

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

## 3.3 Special method names

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

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

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

### 3.3.1 Basic customization

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

Called to create a new instance of class `cls`. `__new__()` is a static method (special-cased so you need not declare it as such) that takes the class of which an instance was requested as its first argument. The remaining arguments are those passed to the object constructor expression (the call to the class). The return value of `__new__()` should be the new object instance (usually an instance of `cls`).

Typical implementations create a new instance of the class by invoking the superclass's `__new__()` method using `super().__new__(cls[, ...])` with appropriate arguments and then modifying the newly created instance as necessary before returning it.

If `__new__()` is invoked during object construction and it returns an instance of `cls`, then the new instance's `__init__()` method will be invoked like `__init__(self[, ...])`, where `self` is the new instance and the remaining arguments are the same as were passed to the object constructor.

If `__new__()` does not return an instance of `cls`, then the new instance's `__init__()` method will not be invoked.

`__new__()` is intended mainly to allow subclasses of immutable types (like `int`, `str`, or `tuple`) to customize instance creation. It is also commonly overridden in custom metaclasses in order to customize class creation.

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

Called after the instance has been created (by `__new__()`), but before it is returned to the caller. The arguments are those passed to the class constructor expression. If a base class has an `__init__()` method, the derived class's `__init__()` method, if any, must explicitly call it to ensure proper initialization of the base class part of the instance; for example: `super().__init__(args...)`.

Because `__new__()` and `__init__()` work together in constructing objects (`__new__()` to create it, and `__init__()` to customize it), no non-`None` value may be returned by `__init__()`; doing so will cause a `TypeError` to be raised at runtime.

`object.__del__(self)`

Called when the instance is about to be destroyed. This is also called a finalizer or (improperly) a destructor. If a base class has a `__del__()` method, the derived class's `__del__()` method, if any, must explicitly call it to ensure proper deletion of the base class part of the instance.

It is possible (though not recommended!) for the `__del__()` method to postpone destruction of the instance by creating a new reference to it. This is called object *resurrection*. It is implementation-dependent whether `__del__()` is called a second time when a resurrected object is about to be destroyed; the current *CPython* implementation only calls it once.

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

#### 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.

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

**Λεπτομέρεια υλοποίησης CPython:** 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. A default implementation is provided by the `object` class itself.

`object.__str__(self)`

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

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

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

`object.__bytes__(self)`

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

`object.__format__(self, format_spec)`

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

See `formatspec` for a description of the standard formatting syntax.

The return value must be a string object.

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

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

Αλλάξε 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()`.

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

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

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

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

```
object.__hash__(self)
```

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

```
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 (inherited from the object class); with them, all objects compare unequal (except with themselves) and `x.__hash__()` returns an appropriate value such that `x == y` implies both that `x is y` and `hash(x) == hash(y)`.

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

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

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

**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://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`.

Αλλάξε 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__()` (which is true of the object class itself), all its instances are considered true.

### 3.3.2 Customizing attribute access

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

`object.__getattr__(self, name)`

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

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

`object.__getattribute__(self, name)`

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

#### 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. An iterable must be returned. `dir()` converts the returned iterable to a list and sorts it.

## Customizing module attribute access

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

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

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

```
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.

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

Added in version 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__`. The object class itself does not implement any of these protocols.

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

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

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

**PEP 252** specifies that `__get__()` is callable with one or two arguments. Python's own built-in descriptors support this specification; however, it is likely that some third-party tools have descriptors that require both



arguments. Python's own `__getattr__()` implementation always passes in both arguments whether they are required or not.

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

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

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

`object.__delete__(self, instance)`

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

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

`object.__objclass__`

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

## Invoking Descriptors

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

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

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

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

### 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

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

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

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



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

## `__slots__`

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

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

### `object.__slots__`

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

Notes on using `__slots__`:

- When inheriting from a class without `__slots__`, the `__dict__` and `__weakref__` attribute of the instances will always be accessible.
- Without a `__dict__` variable, instances cannot be assigned new variables not listed in the `__slots__` definition. Attempts to assign to an unlisted variable name raises `AttributeError`. If dynamic assignment of new variables is desired, then add `'__dict__'` to the sequence of strings in the `__slots__` declaration.
- Without a `__weakref__` variable for each instance, classes defining `__slots__` do not support weak references to its instances. If weak reference support is needed, then add `'__weakref__'` to the sequence of strings in the `__slots__` declaration.
- `__slots__` are implemented at the class level by creating *descriptors* for each variable name. As a result, class attributes cannot be used to set default values for instance variables defined by `__slots__`; otherwise, the class attribute would overwrite the descriptor assignment.
- The action of a `__slots__` declaration is not limited to the class where it is defined. `__slots__` declared in parents are available in child classes. However, instances of a child subclass will get a `__dict__` and `__weakref__` unless the subclass also defines `__slots__` (which should only contain names of any *additional* slots).
- If a class defines a slot also defined in a base class, the instance variable defined by the base class slot is inaccessible (except by retrieving its descriptor directly from the base class). This renders the meaning of the program undefined. In the future, a check may be added to prevent this.
- `TypeError` will be raised if nonempty `__slots__` are defined for a class derived from a "variable-length" built-in type such as `int`, `bytes`, and `tuple`.
- Any non-string *iterable* may be assigned to `__slots__`.
- If a dictionary is used to assign `__slots__`, the dictionary keys will be used as the slot names. The values of the dictionary can be used to provide per-attribute docstrings that will be recognised by `inspect.getdoc()` and displayed in the output of `help()`.
- `__class__` assignment works only if both classes have the same `__slots__`.
- Multiple inheritance with multiple slotted parent classes can be used, but only one parent is allowed to have attributes created by slots (the other bases must have empty slot layouts) - violations raise `TypeError`.
- If an *iterator* is used for `__slots__` then a *descriptor* is created for each of the iterator's values. However, the `__slots__` attribute will be an empty iterator.

## 3.3.3 Customizing class creation

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

**classmethod** object.\_\_init\_subclass\_\_(cls)

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

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

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

Added in version 3.6.

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

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

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

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

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

```
class A:
    pass

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

See *Creating the class object* for more details.

Added in version 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

`object.__mro_entries__(self, bases)`

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

#### Δείτε επίσης

**`types.resolve_bases()`**

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

**`types.get_original_bases()`**

Retrieve a class's «original bases» prior to modifications by `__mro_entries__()`.

**PEP 560**

Core support for typing module and generic types.

### Determining the appropriate metaclass

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

- if no bases and no explicit metaclass are given, then `type()` is used;
- if an explicit metaclass is given and it is *not* an instance of `type()`, then it is used directly as the metaclass;
- if an instance of `type()` is given as the explicit metaclass, or bases are defined, then the most derived metaclass is used.

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

## Preparing the class namespace

Once the appropriate metaclass has been identified, then the class namespace is prepared. If the metaclass has a `__prepare__` attribute, it is called as `namespace = metaclass.__prepare__(name, bases, **kwargs)` (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, **kwargs)` (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.

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

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

- 1) The `type.__new__` method collects all of the attributes in the class namespace that define a `__set_name__()` method;
- 2) Those `__set_name__` methods are called with the class being defined and the assigned name of that particular attribute;
- 3) The `__init_subclass__()` hook is called on the immediate parent of the new class in its method resolution order.

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

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

➔ 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.

`type.__instancecheck__(self, instance)`

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

`type.__subclasscheck__(self, subclass)`

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

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

➔ 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'>
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
>>> 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'>
>>> 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, ...)`. The object class itself does not provide this method.

### 3.3.7 Emulating container types

The following methods can be defined to implement container objects. None of them are provided by the object class itself. Containers usually are *sequences* (such as lists or tuples) or *mappings* (like *dictionaries*), but can represent other containers as well. The first set of methods is used either to emulate a sequence or to emulate a mapping; the difference is that for a sequence, the allowable keys should be the integers  $k$  for which  $0 \leq k < N$  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  $\geq 0$ . 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.

**Λεπτομέρεια υλοποίησης CPython:** 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  $\geq 0$ . The return value may also be `NotImplemented`, which is treated the same as if the `__length_hint__` method didn't exist at all. This method is purely an optimization and is never required for correctness.

Added in version 3.4.

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

#### 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.

`object.__reversed__(self)`

Called (if present) by the `reversed()` built-in to implement reverse iteration. It should return a new iterator object that iterates over all the objects in the container in reverse order.

If the `__reversed__()` method is not provided, the `reversed()` built-in will fall back to using the sequence protocol (`__len__()` and `__getitem__()`). Objects that support the sequence protocol should only provide `__reversed__()` if they can provide an implementation that is more efficient than the one provided by `reversed()`.

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

`object.__contains__(self, item)`

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

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

### 3.3.8 Emulating numeric types

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

`object.__add__(self, other)`

`object.__sub__(self, other)`

`object.__mul__(self, other)`

`object.__matmul__(self, other)`

`object.__truediv__(self, other)`

`object.__floordiv__(self, other)`

`object.__mod__(self, other)`

`object.__divmod__(self, other)`

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

`object.__lshift__(self, other)`

`object.__rshift__(self, other)`

`object.__and__(self, other)`

`object.__xor__(self, other)`

`object.__or__(self, other)`

These methods are called to implement the binary arithmetic operations (+, −, \*, @, /, //, %, `divmod()`, `pow()`, \*\*, <<, >>, &, ^, |). For instance, to evaluate the expression `x + y`, where `x` is an instance of a class that has an `__add__()` method, `type(x).__add__(x, y)` is called. The `__divmod__()` method should be the equivalent to using `__floordiv__()` and `__mod__()`; it should not be related to `__truediv__()`. Note that `__pow__()` should be defined to accept an optional third argument if the three-argument 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 operands are of different types, when the left operand does not support the corresponding operation<sup>3</sup>, or the right operand's class is derived from the left operand's class.<sup>4</sup> For instance, to evaluate the expression `x - y`, where `y` is an instance of a class that has an `__rsub__()` method, `type(y).__rsub__(y, x)` is called if `type(x).__sub__(x, y)` returns `NotImplemented` or `type(y)` is a subclass of `type(x)`.<sup>5</sup>

Note that `__rpow__()` should be defined to accept an optional third argument if the three-argument version of the built-in `pow()` function is to be supported.

Αλλάξε στην έκδοση 3.14: Three-argument `pow()` now try calling `__rpow__()` if necessary. Previously it was only called in two-argument `pow()` and the binary power operator.

#### 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)`

<sup>3</sup> «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.

<sup>4</sup> For operands of the same type, it is assumed that if the non-reflected method (such as `__add__()`) fails then the operation is not supported, which is why the reflected method is not called.

<sup>5</sup> If the right operand's type is a subclass of the left operand's type, the reflected method having precedence allows subclasses to override their ancestors' operations.

```

object.__itruediv__(self, other)
object.__ifloordiv__(self, other)
object.__imod__(self, other)
object.__ipow__(self, other[, modulo])
object.__ilshift__(self, other)
object.__irshift__(self, other)
object.__iand__(self, other)
object.__ixor__(self, other)
object.__ior__(self, other)

```

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

```

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

```

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

```

object.__complex__(self)
object.__int__(self)
object.__float__(self)

```

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

```
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`).

Αλλαξε στην εκδοση 3.14: `int()` no longer delegates to the `__trunc__()` method.

### 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`. The `object` class itself does not provide the context manager methods.

`object.__enter__(self)`

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

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

Exit the runtime context related to this object. The parameters describe the exception that caused the context to be exited. If the context was exited without an exception, all three arguments will be `None`.

If an exception is supplied, and the method wishes to suppress the exception (i.e., prevent it from being propagated), it should return a true value. Otherwise, the exception will be processed normally upon exit from this method.

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

#### ➡ Δείτε επίσης

##### PEP 343 - The «with» statement

The specification, background, and examples for the Python `with` statement.

### 3.3.10 Customizing positional arguments in class pattern matching

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

`object.__match_args__`

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

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

Added in version 3.10.

#### ➡ Δείτε επίσης

##### PEP 634 - Structural Pattern Matching

The specification for the Python `match` statement.

### 3.3.11 Emulating buffer types

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

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

`object.__buffer__(self, flags)`

Called when a buffer is requested from `self` (for example, by the `memoryview` constructor). The `flags` argument is an integer representing the kind of buffer requested, affecting for example whether the returned

buffer is read-only or writable. `inspect.BufferFlags` provides a convenient way to interpret the flags. The method must return a `memoryview` object.

`object.__release_buffer__(self, buffer)`

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

Added in version 3.12.

#### ➡ Δείτε επίσης

##### PEP 688 - Making the buffer protocol accessible in Python

Introduces the Python `__buffer__` and `__release_buffer__` methods.

##### `collections.abc.Buffer`

ABC for buffer types.

### 3.3.12 Annotations

Functions, classes, and modules may contain *annotations*, which are a way to associate information (usually *type hints*) with a symbol.

`object.__annotations__`

This attribute contains the annotations for an object. It is *lazily evaluated*, so accessing the attribute may execute arbitrary code and raise exceptions. If evaluation is successful, the attribute is set to a dictionary mapping from variable names to annotations.

Άλλαξε στην έκδοση 3.14: Annotations are now lazily evaluated.

`object.__annotate__(format)`

An *annotate function*. Returns a new dictionary object mapping attribute/parameter names to their annotation values.

Takes a format parameter specifying the format in which annotations values should be provided. It must be a member of the `annotationlib.Format` enum, or an integer with a value corresponding to a member of the enum.

If an `annotate` function doesn't support the requested format, it must raise `NotImplementedError`. `Annotate` functions must always support `VALUE` format; they must not raise `NotImplementedError()` when called with this format.

When called with `VALUE` format, an `annotate` function may raise `NameError`; it must not raise `NameError` when called requesting any other format.

If an object does not have any annotations, `__annotate__` should preferably be set to `None` (it can't be deleted), rather than set to a function that returns an empty dict.

Added in version 3.14.

#### ➡ Δείτε επίσης

##### PEP 649 — Deferred evaluation of annotation using descriptors

Introduces lazy evaluation of annotations and the `__annotate__` function.

### 3.3.13 Special method lookup

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

```
>>> class C:
...     pass
...
>>> c = C()
>>> c.__len__ = lambda: 5
>>> len(c)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'C' has no len()
```

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

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

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

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

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

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

Bypassing the `__getattr__()` 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()` are also awaitable, but they do not implement `__await__()`.

`object.__await__(self)`

Must return an *iterator*. Should be used to implement *awaitable* objects. For instance, `asyncio.Future` implements this method to be compatible with the *await* expression. The `object` class itself is not awaitable and does not provide this method.

#### Σημείωση

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

Added in version 3.5.

#### 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.

Αλλάξε 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.

Άλλαξε στην έκδοση 3.12: The second signature (type[, value[, traceback]]) is deprecated and may be removed in a future version of Python.

`coroutine.close()`

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

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

### 3.4.3 Asynchronous Iterators

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

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

The object class itself does not provide these methods.

`object.__aiter__(self)`

Must return an *asynchronous iterator* object.

`object.__anext__(self)`

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

An example of an asynchronous iterable object:

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

    def __aiter__(self):
        return self

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

Added in version 3.5.

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.

The object class itself does not provide these methods.

`object.__aenter__(self)`

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



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

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

An example of an asynchronous context manager class:

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

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

Added in version 3.5.



## 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 πού και πώς 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.

Οι παρακάτω δομές δεσμεύουν ονόματα:

- τυπικές παράμετροι συναρτήσεων,
- ορισμοί κλάσεων,
- ορισμοί συναρτήσεων
- εκφράσεις ανάθεσης
- *targets* που είναι αναγνωριστικά αν εμφανίζονται σε μια ανάθεση:
  - επικεφαλίδα βρόχου `for`,
  - μετά το `as` σε μια δήλωση `with`, σε ρήτρα `except`, σε ρήτρα `except*` ή στο `as-pattern` κατά τη δομική αντιστοίχισης μοτίβων,
  - σε ένα στιγμιότυπο μοτίβου κατά τη δομική αντιστοίχισης μοτίβων

- δηλώσεις `import`.
- δηλώσεις `type`.
- *λίστες παραμέτρων τύπου*.

Η δήλωση `import` της μορφής `from ... import *` συνδέει όλα τα ονόματα που ορίζονται στο εισαγόμενο `module`, εκτός από αυτά που ξεκινούν με μια κάτω παύλα. Αυτή η μορφή μπορεί να χρησιμοποιηθεί μόνο στο επίπεδο του `module`.

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`.

Αν μια λειτουργία σύνδεσης ονομάτων συμβεί οπουδήποτε μέσα σε ένα μπλοκ κώδικα, όλες οι χρήσεις του ονόματος μέσα στο μπλοκ αντιμετωπίζονται ως αναφορές στο τρέχον μπλοκ. Αυτό μπορεί να οδηγήσει σε σφάλματα όταν ένα όνομα χρησιμοποιείται μέσα σε ένα μπλοκ πριν δεσμευτεί. Αυτός ο κανόνας είναι λεπτός. Η Python δεν διαθέτει δηλώσεις και επιτρέπει τις λειτουργίες σύνδεσης ονομάτων να συμβαίνουν οπουδήποτε μέσα σε ένα μπλοκ κώδικα. Οι τοπικές μεταβλητές ενός μπλοκ κώδικα μπορούν να προσδιοριστούν σαφώς ολόκληρο το κείμενο του μπλοκ για λειτουργίες σύνδεσης ονομάτων. Δείτε την εγγραφή στο FAQ για το `UnboundLocalError` για παραδείγματα.

Αν η δήλωση `global` εμφανιστεί μέσα σε ένα μπλοκ, όλες οι χρήσεις των ονομάτων που καθορίζονται στη δήλωση αναφέρονται στις συνδέσεις αυτών των ονομάτων στον χώρο ονομάτων ανώτατου επιπέδου. Τα ονόματα επιλύονται στον χώρο ανώτατου επιπέδου αναζητώντας πρώτα στον καθολικό χώρο ονομάτων, δηλαδή τον χώρο ονομάτων του `module` που περιέχει το μπλοκ κώδικα, και στη συνέχεια στο χώρο ονομάτων των `builtins`, τον χώρο ονομάτων του `module builtins`. Ο καθολικός χώρος ονομάτων αναζητείται πρώτος. Αν τα ονόματα δεν βρεθούν εκεί, γίνεται αναζήτηση του ενσωματωμένου χώρου ονομάτων. Εάν τα ονόματα δεν βρίσκονται επίσης στον ενσωματωμένο χώρο ονομάτων, δημιουργούνται νέες μεταβλητές στον καθολικό χώρο ονομάτων. Η καθολική δήλωση πρέπει να προηγείται όλων των χρήσεων των ονομάτων που αναφέρονται.

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.

Η δήλωση `nonlocal` προκαλεί τα αντίστοιχα ονόματα να αναφέρονται σε προηγουμένως δεσμευμένες μεταβλητές στο πλησιέστερο περιβάλλον πεδίου συνάρτησης. Μια εξαίρεση `SyntaxError` εγείρεται κατά το

χρόνο μεταγλώττισης αν το συγκεκριμένο δεν υπάρχει σε κανένα περιβάλλον πεδίου συνάρτησης. Οι *παράμετροι τύπου* δεν μπορούν να δεσμευτούν εκ νέου με τη δήλωση `nonlocal`.

Ο χώρος ονομάτων για ένα `module` δημιουργείται αυτόματα την πρώτη φορά που το `module` εισάγεται. Το κύριο `module` για ένα `script` ονομάζεται πάντα `__main__`.

Τα μπλοκ ορισμού κλάσεων και τα ορίσματα στις συναρτήσεις `exec()` και `eval()` είναι ειδικές περιπτώσεις στο πλαίσιο της επίλυσης ονομάτων. Ένας ορισμός κλάσης είναι μια εκτελέσιμη δήλωση που μπορεί να χρησιμοποιεί και να ορίζει ονόματα. Αυτές οι αναφορές ακολουθούν τους κανονικούς κανόνες επίλυσης ονομάτων, με την εξαίρεση ότι οι αδέσμευτες τοπικές μεταβλητές αναζητούνται στον καθολικό χώρο ονομάτων. Ο χώρος ονομάτων του ορισμού της κλάσης γίνεται το λεξικό χαρακτηριστικών της κλάσης. Το πεδίο των ονομάτων που ορίζονται σε ένα μπλοκ κλάσης περιορίζεται στο μπλοκ της κλάσης· δεν επεκτείνεται στα μπλοκ κώδικα των μεθόδων. Αυτό περιλαμβάνει συνθέσεις και εκφράσεις γεννητριών, αλλά δεν περιλαμβάνει *πεδία σημειώσεων*, τα οποία έχουν πρόσβαση στα περιβάλλοντα πεδία της περιβάλλουσας κλάσης. Αυτό σημαίνει ότι το παρακάτω θα αποτύχει:

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

Ωστόσο, το παρακάτω θα επιτύχει:

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

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

## 4.2.3 Σημειογραφία πεδία

Τα *Annotations*, Οι *λίστες παραμέτρων τύπου* και οι δηλώσεις `type` εισάγουν *πεδία σημειογραφίας*, τα οποία συμπεριφέρονται κυρίως όπως τα πεδία συναρτήσεων, αλλά με κάποιες εξαιρέσεις που συζητούνται παρακάτω.

Τα πεδία σημειογραφίας χρησιμοποιούνται στα παρακάτω συμφραζόμενα:

- *Function annotations*.
- *Variable annotations*.
- Λίστες παραμέτρων τύπου για *generic type aliases*.
- Λίστες τύπου παραμέτρου για *generic functions*. Οι σημειογραφίες μιας γενικής συνάρτησης εκτελούνται μέσα στο πεδίο σημειώσεων, αλλά οι προεπιλογές και οι διακοσμητές της όχι.
- Λίστες παραμέτρων τύπου για *generic classes*. Οι βασικές κλάσεις και τα ορίσματα λέξεων-κλειδιών μιας γενικής κλάσης εκτελούνται μέσα στο πεδίο σημειώσεων, αλλά οι διακοσμητές της όχι.
- Τα όρια, οι περιορισμοί οι προεπιλεγμένες τιμές για παραμέτρους τύπου (*lazily evaluated*).
- Η τιμή των ψευδωνύμων τύπου (*lazily evaluated*).

Τα πεδία σημειογραφίας διαφέρουν από τα πεδία συναρτήσεων με τους εξής τρόπους:

- Τα πεδία σημειογραφίας έχουν πρόσβαση στον χώρο ονομάτων της περιβάλλουσας κλάσης. Αν ένα πεδίο σημειογραφίας βρίσκεται αμέσως μέσα σε έναν χώρο κλάσης ή μέσα σε ένα άλλο πεδίο σημειογραφίας που βρίσκεται αμέσως μέσα σε έναν χώρο κλάσης, ο κώδικας στο πεδίο σημειογραφίας μπορεί να χρησιμοποιεί ονόματα που έχουν οριστεί στον χώρο της κλάσης σαν να εκτελούνταν απευθείας στο σώμα της κλάσης. Αυτό έρχεται σε αντίθεση με τις κανονικές συναρτήσεις που ορίζονται μέσα σε κλάσεις, οι οποίες δεν μπορούν να έχουν πρόσβαση σε ονόματα που έχουν οριστεί στο χώρο της κλάσης.
- Οι εκφράσεις σε πεδία σημειογραφίας δεν μπορούν να περιέχουν τις εκφράσεις `yield`, `yield from`, `await` ή `:=` (python-grammar: assignment\_expression). (Αυτές οι εκφράσεις επιτρέπονται σε άλλα πεδία που περιέχονται μέσα στο πεδίο σημειογραφίας.)

- Τα ονόματα που ορίζονται σε πεδία σημειογραφίας δεν μπορούν να δεσμευτούν εκ νέου με δηλώσεις *nonlocal* σε εσωτερικά πεδία. Αυτό περιλαμβάνει μόνο παραμέτρους τύπου, καθώς και κανένα άλλο συντακτικό στοιχείο που μπορεί να εμφανιστεί μέσα σε πεδία σημειώσεων δεν μπορεί να εισαγάγει νέα ονόματα.
- Ενώ τα πεδία σημειογραφίας έχουν ένα εσωτερικό όνομα, αυτό το όνομα δεν αντικατοπτρίζεται στο *qualified name* των αντικειμένων που ορίζονται μέσα στο πεδίο. Αντίθετα, το `__qualname__` αυτών των αντικειμένων είναι σαν το αντικείμενο να είχε οριστεί στο περιβάλλον πεδίο.

Added in version 3.12: Τα πεδία σημειώσεων εισήχθησαν στην Python 3.12 ως μέρος του **PEP 695**.

Άλλαξε στην έκδοση 3.13: Οι περιοχές σχολίων τύπου χρησιμοποιούνται επίσης για τις προεπιλεγμένες τιμές παραμέτρων τύπου, όπως εισάγεται από το **PEP 696**.

Άλλαξε στην έκδοση 3.14: Οι περιοχές σχολίων τύπου χρησιμοποιούνται επίσης για τα annotations, όπως εισάγεται από το **PEP 649** and **PEP 749**.

## 4.2.4 Καθυστερημένη εκτίμηση

Τα περισσότερα πεδία annotation αξιολογούνται *νωχελικά*. Αυτό περιλαμβάνει annotations, τις τιμές των ψευδωνύμων τύπου που δημιουργούνται μέσω της δήλωσης *type* και τα όρια, τους περιορισμούς και τις προεπιλεγμένες τιμές των μεταβλητών τύπου που δημιουργούνται μέσω της σύνταξης παραμέτρων *type parameter syntax*. Αυτό σημαίνει ότι δεν αξιολογούνται όταν δημιουργείται το ψευδώνυμο τύπου ή η μεταβλητή τύπου ή όταν δημιουργείται το αντικείμενο που φέρει σχολιασμούς. Αντίθετα, αξιολογούνται μόνο όταν είναι απαραίτητο, για παράδειγμα όταν γίνεται πρόσβαση στο χαρακτηριστικό `__value__` σε ένα ψευδώνυμο τύπου.

Παράδειγμα:

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

Εδώ η εξαίρεση εγείρεται μόνο όταν γίνει πρόσβαση στο χαρακτηριστικό `__value__` του ψευδωνύμου τύπου ή στο χαρακτηριστικό `__bound__` της μεταβλητής τύπου.

Αυτή η συμπεριφορά είναι κυρίως χρήσιμη για αναφορές σε τύπους που δεν έχουν ακόμη οριστεί κατά τη δημιουργία του ψευδωνύμου τύπου ή της μεταβλητής τύπου. Για παράδειγμα, η καθυστερημένη εκτίμηση επιτρέπει τη δημιουργία αμοιβαίων αναδρομικών ψευδωνύμων τύπων:

```
from typing import Literal

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

Οι τιμές που αξιολογούνται καθυστερημένα αξιολογούνται στο *πεδίο σημειογραφίας*, που σημαίνει ότι τα ονόματα που εμφανίζονται μέσα στην καθυστερημένα αξιολογούμενη τιμή αναζητούνται σαν να χρησιμοποιήθηκαν στο αμέσως περιβάλλον πεδίο.

Added in version 3.12.

## 4.2.5 Ενσωματωμένες συναρτήσεις και περιορισμένη εκτέλεση

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`.

## 4.2.6 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.<sup>1</sup> 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`.

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.

<sup>1</sup> Αυτός ο περιορισμός προκύπτει επειδή ο κώδικας που εκτελείται από αυτές τις λειτουργίες δεν είναι διαθέσιμος τη στιγμή που το module μεταγλωττίζεται.

**i 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 διεργμνέα.

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

## Υποσημειώσεις



---

## 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<sup>1</sup>, 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.

Αλλαξε 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**).

### 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.

---

<sup>1</sup> See `types.ModuleType`.

## 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.

Αλλαξε 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*).

Αλλάξε 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()`.

Αλλάξε στην έκδοση 3.10: Use of `find_module()` by the import system now raises `ImportWarning`.

Αλλάξε στην έκδοση 3.12: `find_module()` has been removed. Use `find_spec()` instead.

## 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:
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

    # unsupported
    raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    # namespace package
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
else:
    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`<sup>2</sup>.

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.

<sup>2</sup> 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.

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.

Added in version 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`.
- 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.

Άλλαξε στην έκδοση 3.10: Use of `load_module()` will raise `ImportWarning`.

## 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 specs

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 `module.__spec__`. 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*.

See `ModuleSpec` for details on the contents of the module spec.

Added in version 3.4.

### 5.4.4 `__path__` attributes on modules

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

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

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

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

### 5.4.5 Module reprs

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

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

Here are the exact rules used:

- If the module has a `__spec__` attribute, the information in the spec is used to generate the repr. The «name», «loader», «origin», and «has\_location» attributes are consulted.
- If the module has a `__file__` attribute, this is used as part of the module's repr.
- If the module has no `__file__` but does have a `__loader__` that is not `None`, then the loader's repr is used as part of the module's repr.
- Otherwise, just use the module's `__name__` in the repr.

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

### 5.4.6 Cached bytecode invalidation

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

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

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 should be present on `sys.path`; all other data types are ignored.

The *path based finder* is a *meta path finder*, so the import machinery begins the *import path* search by calling the path based finder's `find_spec()` method as described previously. When the `path` argument to `find_spec()`



is given, it will be a list of string paths to traverse - typically a package's `__path__` attribute for an import within that package. If the `path` argument is `None`, this indicates a top level import and `sys.path` is used.

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

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

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

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

The current working directory - denoted by an empty string - is handled slightly differently from other entries on `sys.path`. First, if the current working directory cannot be determined or is found not to 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.

Αλλάξε 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).

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()`.

Άλλαξε στην έκδοση 3.10: Calls to `find_module()` and `find_loader()` by the import system will raise `ImportWarning`.

Άλλαξε στην έκδοση 3.12: `find_module()` and `find_loader()` have been removed.

## 5.6 Replacing the standard import system

The most reliable mechanism for replacing the entire import system is to delete the default contents of `sys.meta_path`, replacing them entirely with a custom meta path hook.

If it is acceptable to only alter the behaviour of import statements without affecting other APIs that access the import system, then replacing the builtin `__import__()` function may be sufficient. This technique may also be employed at the module level to only alter the behaviour of import statements within that module.

To selectively prevent the import of some modules from a hook early on the meta path (rather than disabling the standard import system entirely), it is sufficient to raise `ModuleNotFoundError` directly from `find_spec()` instead of returning `None`. The latter indicates that the meta path search should continue, while raising an exception terminates it immediately.

## 5.7 Package Relative Imports

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

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

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

```
from .moduleY import spam
from .moduleY import spam as ham
from . import moduleY
from ..subpackage1 import moduleY
from ..subpackage2.moduleZ import eggs
from ..moduleA import foo
```

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

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

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

## 5.8 Special considerations for `__main__`

The `__main__` module is a special case relative to Python's import system. As noted *elsewhere*, the `__main__` module is directly initialized at interpreter startup, much like `sys` and `builtins`. However, unlike those two, it

doesn't strictly qualify as a built-in module. This is because the manner in which `__main__` is initialized depends on the flags and other options with which the interpreter is invoked.

### 5.8.1 `__main__`.`__spec__`

Depending on how `__main__` is initialized, `__main__.__spec__` gets set appropriately or to `None`.

When Python is started with the `-m` option, `__spec__` is set to the module spec of the corresponding module or package. `__spec__` is also populated when the `__main__` module is loaded as part of executing a directory, zipfile or other `sys.path` entry.

In the remaining cases `__main__.__spec__` is set to `None`, as the code used to populate the `__main__` does not correspond directly with an importable module:

- interactive prompt
- `-c` 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 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.



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 real type», this means that the operator implementation for built-in types works as follows:

- If both arguments are complex numbers, no conversion is performed;
- if either argument is a complex or a floating-point number, the other is converted to a floating-point number;
- 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 *Names (identifiers and keywords)* for lexical definition and section *Όνομασία 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.

### ➡ Δείτε επίσης

The *class specifications*.

More precisely, private names are transformed to a longer form before code is generated for them. If the transformed name is longer than 255 characters, implementation-defined truncation may happen.

The transformation is independent of the syntactical context in which the identifier is used but only the following private identifiers are mangled:

- Any name used as the name of a variable that is assigned or read or any name of an attribute being accessed. The `__name__` attribute of nested functions, classes, and type aliases is however not mangled.
- The name of imported modules, e.g., `__spam` in `import __spam`. If the module is part of a package (i.e., its name contains a dot), the name is *not* mangled, e.g., the `__foo` in `import __foo.bar` is not mangled.
- The name of an imported member, e.g., `__f` in `from spam import __f`.

The transformation rule is defined as follows:

- The class name, with leading underscores removed and a single leading underscore inserted, is inserted in front of the identifier, e.g., the identifier `__spam` occurring in a class named `Foo`, `_Foo` or `__Foo` is transformed to `_Foo__spam`.
- If the class name consists only of underscores, the transformation is the identity, e.g., the identifier `__spam` occurring in a class named `_` or `__` is left as is.

## 6.2.2 Literals

Python supports string and bytes literals and various numeric literals:

```
literal: strings | NUMBER
```

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. See section *String literal concatenation* for details on *strings*.

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.

### 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:

```
>>> "hello" 'world'
"helloworld"
```

Formally:

```
strings: ( STRING | fstring)+ | tstring+
```

This feature is defined at the syntactical level, so it only works with literals. To concatenate string expressions at run time, the `+` operator may be used:

```
>>> greeting = "Hello"
>>> space = " "
>>> name = "Blaise"
>>> print(greeting + space + name)    # not: print(greeting space name)
Hello Blaise
```

Literal concatenation can freely mix raw strings, triple-quoted strings, and formatted string literals. For example:

```
>>> "Hello" r', ' f"{name}!"
"Hello, Blaise!"
```

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
           )
```

However, bytes literals may only be combined with other byte literals; not with string literals of any kind. Also, template string literals may only be combined with other template string literals:

```
>>> t"Hello" t"{name}!"
Template(strings=('Hello', '!'), interpolations=())
```

## 6.2.3 Parenthesized forms

A parenthesized form is an optional expression list enclosed in parentheses:

```
parenth_form: "(" [starred_expression] ")"
```

A parenthesized expression list yields whatever that expression list yields: if the list contains at least one comma, it yields a tuple; otherwise, it yields the single expression that makes up the expression list.

An empty pair of parentheses yields an empty tuple object. Since tuples are immutable, the same rules as for literals apply (i.e., two occurrences of the empty tuple may or may not yield the same object).

Note that tuples are not formed by the parentheses, but rather by use of the comma. The exception is the empty tuple, for which parentheses *are* required — allowing unparenthesized «nothing» in expressions would cause ambiguities and allow common typos to pass uncaught.

## 6.2.4 Displays for lists, sets and dictionaries

For constructing a list, a set or a dictionary Python provides special syntax called «displays», each of them in two flavors:

- either the container contents are listed explicitly, or
- they are computed via a set of looping and filtering instructions, called a *comprehension*.

Common syntax elements for comprehensions are:

```
comprehension: assignment_expression comp_for
comp_for:      ["async"] "for" target_list "in" or_test [comp_iter]
comp_iter:     comp_for | comp_if
comp_if:       "if" or_test [comp_iter]
```

The comprehension consists of a single expression followed by at least one `for` clause and zero or more `for` or `if` clauses. In this case, the elements of the new container are those that would be produced by considering each of the `for` or `if` clauses a block, nesting from left to right, and evaluating the expression to produce an element each time the innermost block is reached.

However, aside from the iterable expression in the leftmost `for` clause, the comprehension is executed in a separate implicitly nested scope. This ensures that names assigned to in the target list don't «leak» into the enclosing scope.

The iterable expression in the leftmost `for` clause is evaluated directly in the enclosing scope and then passed as an argument to the implicitly nested scope. Subsequent `for` clauses and any filter condition in the leftmost `for` clause cannot be evaluated in the enclosing scope as they may depend on the values obtained from the leftmost iterable. For example: `[x*y for x in range(10) for y in range(x, x+10)]`.

To ensure the comprehension always results in a container of the appropriate type, `yield` and `yield from` expressions are prohibited in the implicitly nested scope.

Since Python 3.6, in an `async def` function, an `async for` clause may be used to iterate over a *asynchronous iterator*. A comprehension in an `async def` function may consist of either a `for` or `async for` clause following the leading expression, may contain additional `for` or `async for` clauses, and may also use `await` expressions.

If a comprehension contains `async for` clauses, or if it contains `await` expressions or other asynchronous comprehensions anywhere except the iterable expression in the leftmost `for` clause, it is called an *asynchronous comprehension*. An asynchronous comprehension may suspend the execution of the coroutine function in which it appears. See also [PEP 530](#).

Added in version 3.6: Asynchronous comprehensions were introduced.

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

Άλλαξε στην έκδοση 3.11: Asynchronous comprehensions are now allowed inside comprehensions in asynchronous functions. Outer comprehensions implicitly become asynchronous.

## 6.2.5 List displays

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

```
list_display: "[" [flexible_expression_list | comprehension] "]"
```

A list display yields a new list object, the contents being specified by either a list of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and placed into the list object in that order. When a comprehension is supplied, the list is constructed from the elements resulting from the comprehension.

## 6.2.6 Set displays

A set display is denoted by curly braces and distinguishable from dictionary displays by the lack of colons separating keys and values:

```
set_display: "{" (flexible_expression_list | comprehension) "}"
```

A set display yields a new mutable set object, the contents being specified by either a sequence of expressions or a comprehension. When a comma-separated list of expressions is supplied, its elements are evaluated from left to right and added to the set object. When a comprehension is supplied, the set is constructed from the elements resulting from the comprehension.

An empty set cannot be constructed with `{ }`; this literal constructs an empty dictionary.

## 6.2.7 Dictionary displays

A dictionary display is a possibly empty series of dict items (key/value pairs) enclosed in curly braces:

```
dict_display: "{" [dict_item_list | dict_comprehension] "}"
dict_item_list: dict_item ("," dict_item)* [","]
dict_item: expression ":" expression | "***" or_expr
dict_comprehension: expression ":" expression comp_for
```

A dictionary display yields a new dictionary object.

If a comma-separated sequence of dict items is given, they are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding value. This means that you



can specify the same key multiple times in the dict item list, and the final dictionary's value for that key will be the last one given.

A double asterisk `**` denotes *dictionary unpacking*. Its operand must be a *mapping*. Each mapping item is added to the new dictionary. Later values replace values already set by earlier dict items and earlier dictionary unpackings.

Added in version 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 value (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, and the *iterator* is immediately created for that iterable, so that an error produced while creating the iterator 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](#)).

Added in version 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_from:          "yield" "from" expression
yield_expression:    "yield" yield_list | yield_from
```

The `yield` expression is used when defining a *generator* function or an *asynchronous generator* function and thus can only be used in the body of a function definition. Using a `yield` expression in a function's body causes that function to be a generator function, and using it in an `async def` function's body causes that coroutine function to be an asynchronous generator function. For example:

```
def gen(): # defines a generator function
    yield 123

async def agen(): # defines an asynchronous generator function
    yield 123
```

Due to their side effects on the containing scope, `yield` expressions are not permitted as part of the implicitly defined scopes used to implement comprehensions and generator expressions.

Αλλαξε 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 `yield_list` to the generator's caller, or `None` if `yield_list` is omitted. By suspended, we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by calling one of the generator's methods, the function can proceed exactly as if the `yield` expression were just another external call. The value of the `yield` expression after resuming depends on the method which resumed the execution. If `__next__()` is used (typically via either a `for` or the `next()` builtin) then the result is `None`. Otherwise, if `send()` is used, then the result will be the value passed in to that method.

All of this makes generator functions quite similar to coroutines; they yield multiple times, they have more than one entry point and their execution can be suspended. The only difference is that a generator function cannot control where the execution should continue after it yields; the control is always transferred to the generator's caller.

Yield expressions are allowed anywhere in a `try` construct. If the generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), the generator-iterator's `close()` method will be called, allowing any pending `finally` clauses to execute.

When `yield from <expr>` is used, the supplied expression must be an iterable. The values produced by iterating that iterable are passed directly to the caller of the current generator's methods. Any values passed in with `send()` and any exceptions passed in with `throw()` are passed to the underlying iterator if it has the appropriate methods. If this is not the case, then `send()` will raise `AttributeError` or `TypeError`, while `throw()` will just raise the passed in exception immediately.

When the underlying iterator is complete, the `value` attribute of the raised `StopIteration` instance becomes the value of the `yield` expression. It can be either set explicitly when raising `StopIteration`, or automatically when the subiterator is a generator (by returning a value from the subgenerator).

Αλλαξε 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 `yield_list` is returned to `__next__()`'s caller. If the generator exits without yielding another value, a `StopIteration` exception is raised.

This method is normally called implicitly, e.g. by a `for` loop, or by the built-in `next()` function.

`generator.send(value)`

Resumes the execution and «sends» a value into the generator function. The `value` argument becomes the result of the current `yield` expression. The `send()` method returns the next value yielded by the generator, or raises `StopIteration` if the generator exits without yielding another value. When `send()` is called to start the generator, it must be called with `None` as the argument, because there is no `yield` expression that could receive the value.

`generator.throw(value)`

`generator.throw(type[, value[, traceback]])`

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

In typical use, this is called with a single exception instance similar to the way the `raise` keyword is used.

For backwards compatibility, however, the second signature is supported, following a convention from older versions of Python. The `type` argument should be an exception class, and `value` should be an exception instance. If the `value` is not provided, the `type` constructor is called to get an instance. If `traceback` is provided, it is set on the exception, otherwise any existing `__traceback__` attribute stored in `value` may be cleared.

Αλλάξε στην έκδοση 3.12: The second signature (`type[, value[, traceback]]`) is deprecated and may be removed in a future version of Python.

`generator.close()`

Raises a `GeneratorExit` exception at the point where the generator function was paused (equivalent to calling `throw(GeneratorExit)`). The exception is raised by the `yield` expression where the generator was paused. If the generator function catches the exception and returns a value, this value is returned from `close()`. If the generator function is already closed, or raises `GeneratorExit` (by not catching the exception), `close()` returns `None`. If the generator yields a value, a `RuntimeError` is raised. If the generator raises any other exception, it is propagated to the caller. If the generator has already exited due to an exception or normal exit, `close()` returns `None` and has no other effect.

Αλλάξε στην έκδοση 3.13: If a generator returns a value upon being closed, the value is returned by `close()`.

## 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:
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

...         value = (yield value)
...     except Exception as e:
...         value = e
...     finally:
...         print("Don't forget to clean up when 'close()' is called.")
...
>>> generator = echo(1)
>>> print(next(generator))
Execution starts when 'next()' is called for the first time.
1
>>> print(next(generator))
None
>>> print(generator.send(2))
2
>>> generator.throw(TypeError, "spam")
TypeError('spam',)
>>> generator.close()
Don't forget to clean up when 'close()' is called.

```

For examples using `yield from`, see pep-380 in «What's New in Python.»

### 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 `yield_list` to the awaiting coroutine. As with a generator, suspension means that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by awaiting on the next object returned by the asynchronous generator's methods, the function can proceed exactly as if the `yield` expression were just another external call. The value of the `yield` expression after resuming depends on the method which resumed the execution. If `__anext__()` is used then the result is `None`. Otherwise, if `asend()` is used, then the result will be the value passed in to that method.

If an asynchronous generator happens to exit early by `break`, the caller task being cancelled, or other exceptions, the generator's `async` cleanup code will run and possibly raise exceptions or access context variables in an unexpected context—perhaps after the lifetime of tasks it depends, or during the event loop shutdown when the `async-generator` garbage collection hook is called. To prevent this, the caller must explicitly close the `async` generator by calling `aclose()` method to finalize the generator and ultimately detach it from the event loop.

In an asynchronous generator function, `yield` expressions are allowed anywhere in a `try` construct. However, if an asynchronous generator is not resumed before it is finalized (by reaching a zero reference count or by being garbage collected), then a `yield` expression within a `try` construct could result in a failure to execute pending *finally* clauses. In this case, it is the responsibility of the event loop or scheduler running the asynchronous generator to call the asynchronous generator-iterator's `aclose()` method and run the resulting coroutine object, thus allowing any pending *finally* clauses to execute.

To take care of finalization upon event loop termination, an event loop should define a *finalizer* function which takes an asynchronous generator-iterator and presumably calls `aclose()` and executes the coroutine. This *finalizer* may be registered by calling `sys.set_asyncgen_hooks()`. When first iterated over, an asynchronous generator-iterator will store the registered *finalizer* to be called upon finalization. For a reference example of a *finalizer* method see the implementation of `asyncio.Loop.shutdown_asyncgens` in `Lib/asyncio/base_events.py`.

The expression `yield from <expr>` is a syntax error when used in an asynchronous generator function.

### Asynchronous generator-iterator methods

This subsection describes the methods of an asynchronous generator iterator, which are used to control the execution of a generator function.

**async** `agen.__anext__()`

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

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

**async** `agen.asend(value)`

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

**async** `agen.athrow(value)`

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

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

Άλλαξε στην έκδοση 3.12: The second signature (`type[, value[, traceback]]`) is deprecated and may be removed in a future version of Python.

**async** `agen.aclose()`

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

## 6.3 Primaries

Primaries represent the most tightly bound operations of the language. Their syntax is:

```
primary: atom | attributeref | subscription | slicing | call
```

### 6.3.1 Attribute references

An attribute reference is a primary followed by a period and a name:

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

The primary must evaluate to an object of a type that supports attribute references, which most objects do. This object is then asked to produce the attribute whose name is the identifier. The type and value produced is determined by the object. Multiple evaluations of the same attribute reference may yield different objects.

This production can be customized by overriding the `__getattribute__()` method or the `__getattr__()` method. The `__getattribute__()` method is called first and either returns a value or raises `AttributeError` if the attribute is not available.

If an `AttributeError` is raised and the object has a `__getattr__()` method, that method is called as a fallback.

### 6.3.2 Subscriptions

The subscription of an instance of a *container class* will generally select an element from the container. The subscription of a *generic class* will generally return a `GenericAlias` object.

```
subscription: primary "[" flexible_expression_list "]"
```

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

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

If the expression list contains at least one comma, or if any of the expressions are starred, the expression list will evaluate to a tuple containing the items of the expression list. Otherwise, the expression list will evaluate to the value of the list's sole member.

Άλλαξε στην έκδοση 3.11: Expressions in an expression list may be starred. See [PEP 646](#).

For built-in objects, there are two types of objects that support subscription via `__getitem__()`:

1. Mappings. If the primary is a *mapping*, the expression list must evaluate to an object whose value is one of the keys of the mapping, and the subscription selects the value in the mapping that corresponds to that key. An example of a builtin mapping class is the `dict` class.
2. Sequences. If the primary is a *sequence*, the expression list must evaluate to an `int` or a `slice` (as discussed in the following section). Examples of builtin sequence classes include the `str`, `list` and `tuple` classes.

The formal syntax makes no special provision for negative indices in *sequences*. However, built-in sequences all provide a `__getitem__()` method that interprets negative indices by adding the length of the sequence to the index so that, for example, `x[-1]` selects the last item of `x`. The resulting value must be a nonnegative integer less than the number of items in the sequence, and the subscription selects the item whose index is that value (counting from zero). Since the support for negative indices and slicing occurs in the object's `__getitem__()` method, subclasses overriding this method will need to explicitly add that support.

A `string` is a special kind of sequence whose items are *characters*. A character is not a separate data type but a string of exactly one character.

### 6.3.3 Slicings

A slicing selects a range of items in a sequence object (e.g., a string, tuple or list). Slicings may be used as expressions or as targets in assignment or *del* statements. The syntax for a slicing:

```
slicing:      primary "[" slice_list "]"
slice_list:   slice_item ("," slice_item) * [" , "]
slice_item:   expression | proper_slice
proper_slice: [lower_bound] ":" [upper_bound] [ ":" [stride] ]
```



```

lower_bound:  expression
upper_bound:  expression
stride:       expression

```

There is ambiguity in the formal syntax here: anything that looks like an expression list also looks like a slice list, so any subscription can be interpreted as a slicing. Rather than further complicating the syntax, this is disambiguated by defining that in this case the interpretation as a subscription takes priority over the interpretation as a slicing (this is the case if the slice list contains no proper slice).

The semantics for a slicing are as follows. The primary is indexed (using the same `__getitem__()` method as normal subscription) with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of a proper slice is a slice object (see section *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 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.

**Λεπτομέρεια υλοποίησης CPython:** An implementation may provide built-in functions whose positional parameters do not have names, even if they are “named” for the purpose of documentation, and which therefore cannot be supplied by keyword. In CPython, this is the case for functions implemented in C that use `PyArg_ParseTuple()` to parse their arguments.

If there are more positional arguments than there are formal parameter slots, a `TypeError` exception is raised, unless a formal parameter using the syntax `*identifier` is present; in this case, that formal parameter receives a tuple containing the excess positional arguments (or an empty tuple if there were no excess positional arguments).

If any keyword argument does not correspond to a formal parameter name, a `TypeError` exception is raised, unless a formal parameter using the syntax `**identifier` is present; in this case, that formal parameter receives a dictionary containing the excess keyword arguments (using the keywords as keys and the argument values as corresponding values), or a (new) empty dictionary if there were no excess keyword arguments.

If the syntax `*expression` appears in the function call, `expression` must evaluate to an *iterable*. Elements from these iterables are treated as if they were additional positional arguments. For the call `f(x1, x2, *y, x3, x4)`, if `y` evaluates to a sequence `y1, ..., yM`, this is equivalent to a call with `M+4` positional arguments `x1, x2, y1, ..., yM, x3, x4`.

A consequence of this is that although the `*expression` syntax may appear *after* explicit keyword arguments, it is processed *before* the keyword arguments (and any `**expression` arguments – see below). So:

```
>>> def f(a, b):
...     print(a, b)
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

It is unusual for both keyword arguments and the `*expression` syntax to be used in the same call, so in practice this confusion does not often arise.

If the syntax `**expression` appears in the function call, `expression` must evaluate to a *mapping*, the contents of which are treated as additional keyword arguments. If a parameter matching a key has already been given a value (by an explicit keyword argument, or from another unpacking), a `TypeError` exception is raised.

When `**expression` is used, each key in this mapping must be a string. Each value from the mapping is assigned to the first formal parameter eligible for keyword assignment whose name is equal to the key. A key need not be a Python identifier (e.g. `"max-temp °F"` is acceptable, although it will not match any formal parameter that could be declared). If there is no match to a formal parameter the key-value pair is collected by the `**` parameter, if there is one, or if there is not, a `TypeError` exception is raised.

Formal parameters using the syntax `*identifier` or `**identifier` cannot be used as positional argument slots or as keyword argument names.

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. If execution reaches the end of the code block without executing a `return` statement, the return value is `None`.

#### a built-in function or method:

The result is up to the interpreter; see built-in-funcs for the descriptions of built-in functions and methods.

#### a class object:

A new instance of that class is returned.

#### a class instance method:

The corresponding user-defined function is called, with an argument list that is one longer than the argument



list of the call: the instance becomes the first argument.

#### a class instance:

The class must define a `__call__()` method; the effect is then the same as if that method was called.

## 6.4 Await expression

Suspend the execution of *coroutine* on an *awaitable* object. Can only be used inside a *coroutine function*.

```
await_expr: "await" primary
```

Added in version 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__()` and `__rpow__()` methods.

## 6.6 Unary arithmetic and bitwise operations

All unary arithmetic and bitwise operations have the same priority:

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

The unary `-` (minus) operator yields the negation of its numeric argument; the operation can be overridden with the `__neg__()` special method.

The unary `+` (plus) operator yields its numeric argument unchanged; the operation can be overridden with the `__pos__()` special method.

The unary `~` (invert) operator yields the bitwise inversion of its integer argument. The bitwise inversion of `x` is defined as  $-(x+1)$ . It only applies to integral numbers or to custom objects that override the `__invert__()` special method.

In all three cases, if the argument does not have the proper type, a `TypeError` exception is raised.

## 6.7 Binary arithmetic operations

The binary arithmetic operations have the conventional priority levels. Note that some of these operations also apply to certain non-numeric types. Apart from the power operator, there are only two levels, one for multiplicative operators and one for additive operators:

```
m_expr: u_expr | m_expr "*" u_expr | m_expr "@" m_expr |
```

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

The `*` (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer and the other must be a sequence. In the former case, the numbers are converted to a common real 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.

Άλλαξε στην έκδοση 3.14: If only one operand is a complex number, the other operand is converted to a floating-point number.

The `@` (at) operator is intended to be used for matrix multiplication. No builtin Python types implement this operator.

This operation can be customized using the special `__matmul__()` and `__rmatmul__()` methods.

Added in version 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.

The division operation can be customized using the special `__truediv__()` and `__rtruediv__()` methods. The floor division operation can be customized using the special `__floordiv__()` and `__rfloordiv__()` methods.

The `%` (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the `ZeroDivisionError` exception. The arguments may be floating-point numbers, e.g., `3.14%0.7` equals `0.34` (since `3.14` equals `4*0.7 + 0.34`.) The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand<sup>1</sup>.

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)`<sup>2</sup>.

In addition to performing the modulo operation on numbers, the `%` operator is also overloaded by string objects to perform old-style string formatting (also known as interpolation). The syntax for string formatting is described in the Python Library Reference, section `old-string-formatting`.

The *modulo* operation can be customized using the special `__mod__()` and `__rmod__()` methods.

The floor division operator, the modulo operator, and the `divmod()` function are not defined for complex numbers. Instead, convert to a floating-point number using the `abs()` function if appropriate.

The `+` (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both be sequences of the same type. In the former case, the numbers are converted to a common real 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.

Άλλαξε στην έκδοση 3.14: If only one operand is a complex number, the other operand is converted to a floating-point number.

The `-` (subtraction) operator yields the difference of its arguments. The numeric arguments are first converted to a common real type.

This operation can be customized using the special `__sub__()` and `__rsub__()` methods.

---

<sup>1</sup> While `abs(x%y) < abs(y)` is true mathematically, for floats it may not be true numerically due to roundoff. For example, and assuming a platform on which a Python float is an IEEE 754 double-precision number, in order that `-1e-100 % 1e100` have the same sign as `1e100`, the computed result is `-1e-100 + 1e100`, which is numerically exactly equal to `1e100`. The function `math.fmod()` returns a result whose sign matches the sign of the first argument instead, and so returns `-1e-100` in this case. Which approach is more appropriate depends on the application.

<sup>2</sup> 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`.

Άλλαξε στην έκδοση 3.14: If only one operand is a complex number, the other operand is converted to a floating-point number.

## 6.8 Shifting operations

The shifting operations have lower priority than the arithmetic operations:

```
shift_expr: a_expr | shift_expr ("<<" | ">>") a_expr
```

These operators accept integers as arguments. They shift the first argument to the left or right by the number of bits given by the second argument.

The left shift operation can be customized using the special `__lshift__()` and `__rlshift__()` methods. The right shift operation can be customized using the special `__rshift__()` and `__rrshift__()` methods.

A right shift by  $n$  bits is defined as floor division by `pow(2, n)`. A left shift by  $n$  bits is defined as multiplication with `pow(2, n)`.

## 6.9 Binary bitwise operations

Each of the three bitwise operations has a different priority level:

```
and_expr: shift_expr | and_expr "&" shift_expr
xor_expr: and_expr | xor_expr "^" and_expr
or_expr:  xor_expr | or_expr "|" xor_expr
```

The `&` operator yields the bitwise AND of its arguments, which must be integers or one of them must be a custom object overriding `__and__()` or `__rand__()` special methods.

The `^` operator yields the bitwise XOR (exclusive OR) of its arguments, which must be integers or one of them must be a custom object overriding `__xor__()` or `__rxor__()` special methods.

The `|` operator yields the bitwise (inclusive) OR of its arguments, which must be integers or one of them must be a custom object overriding `__or__()` or `__ror__()` special methods.

## 6.10 Comparisons

Unlike C, all comparison operations in Python have the same priority, which is lower than that of any arithmetic, shifting or bitwise operation. Also unlike C, expressions like `a < b < c` have the interpretation that is conventional in mathematics:

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

Comparisons yield boolean values: `True` or `False`. Custom *rich comparison methods* may return non-boolean values. In this case Python will call `bool()` on such value in boolean contexts.

Comparisons can be chained arbitrarily, e.g., `x < y <= z` is equivalent to `x < y` and `y <= z`, except that `y` is evaluated only once (but in both cases `z` is not evaluated at all when `x < y` is found to be false).

Formally, if `a, b, c, ..., y, z` are expressions and `op1, op2, ..., opN` are comparison operators, then `a op1 b op2 c ... y opN z` is equivalent to `a op1 b` and `b op2 c` and `... y opN z`, except that each expression is evaluated at most once.

Note that `a op1 b op2 c` doesn't imply any kind of comparison between `a` and `c`, so that, e.g., `x < y > z` is perfectly legal (though perhaps not pretty).

### 6.10.1 Value comparisons

The operators `<`, `>`, `==`, `>=`, `<=`, and `!=` compare the values of two objects. The objects do not need to have the same type.

Chapter *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 (types `numeric`) and of the standard library types `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.<sup>3</sup>

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.

---

<sup>3</sup> 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()`.

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.

- 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.<sup>4</sup>

## 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*.

---

<sup>4</sup> 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.

One common use case is when handling matched regular expressions:

```
if matching := pattern.search(data):
    do_something(matching)
```

Or, when processing a file stream in chunks:

```
while chunk := file.read(9000):
    process(chunk)
```

Assignment expressions must be surrounded by parentheses when used as expression statements and when used as sub-expressions in slicing, conditional, lambda, keyword-argument, and comprehension-if expressions and in `assert`, `with`, and assignment statements. In all other places where they can be used, parentheses are not required, including in `if` and `while` statements.

Added in version 3.8: 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

```
starred_expression:      "*" or_expr | expression
flexible_expression:     assignment_expression | starred_expression
flexible_expression_list: flexible_expression ("," flexible_expression)* [","]
starred_expression_list: starred_expression ("," starred_expression)* [","]
expression_list:         expression ("," expression)* [","]
yield_list:              expression_list | starred_expression "," [starred_expression_list]
```

Except when part of a list or set display, an expression list containing at least one comma yields a tuple. The length of the tuple is the number of expressions in the list. The expressions are evaluated from left to right.

An asterisk `*` denotes *iterable unpacking*. Its operand must be an *iterable*. The iterable is expanded into a sequence of items, which are included in the new tuple, list, or set, at the site of the unpacking.

Added in version 3.5: Iterable unpacking in expression lists, originally proposed by [PEP 448](#).



Added in version 3.11: Any item in an expression list may be starred. See [PEP 646](#).

A trailing comma is required only to create a one-item tuple, such as `1, ;` it is optional in all other cases. A single expression without a trailing comma doesn't create a tuple, but rather yields the value of that expression. (To create an empty tuple, use an empty pair of parentheses: `()`.)

## 6.16 Evaluation order

Python evaluates expressions from left to right. Notice that while evaluating an assignment, the right-hand side is evaluated before the left-hand side.

In the following lines, expressions will be evaluated in the arithmetic order of their suffixes:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
expr1(expr2, expr3, *expr4, **expr5)
expr3, expr4 = expr1, expr2
```

## 6.17 Operator precedence

The following table summarizes the operator precedence in Python, from highest precedence (most binding) to lowest precedence (least binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation and conditional expressions, which group from right to left).

Note that comparisons, membership tests, and identity tests, all have the same precedence and have a left-to-right chaining feature as described in the [Comparisons](#) section.

Operator	Description
<code>(expressions...),</code> <code>[expressions...], {key: value...},</code> <code>{expressions...}</code>	Binding or parenthesized expression, list display, dictionary display, set display
<code>x[index], x[index:index], x(arguments...),</code> <code>x.attribute</code>	Subscription, slicing, call, attribute reference
<code>await x</code>	Await expression
<code>**</code>	Exponentiation <sup>5</sup>
<code>+x, -x, ~x</code>	Positive, negative, bitwise NOT
<code>*, @, /, //, %</code>	Multiplication, matrix multiplication, division, floor division, remainder <sup>6</sup>
<code>+, -</code>	Addition and subtraction
<code>&lt;&lt;, &gt;&gt;</code>	Shifts
<code>&amp;</code>	Bitwise AND
<code>^</code>	Bitwise XOR
<code> </code>	Bitwise OR
<code>in, not in, is, is not, &lt;, &lt;=, &gt;, &gt;=, !=, ==</code>	Comparisons, including membership tests and identity tests
<code>not x</code>	Boolean NOT
<code>and</code>	Boolean AND
<code>or</code>	Boolean OR
<code>if-else</code>	Conditional expression
<code>lambda</code>	Lambda expression
<code>:=</code>	Assignment expression

<sup>5</sup> The power operator `**` binds less tightly than an arithmetic or bitwise unary operator on its right, that is, `2**~1` is `0.5`.

<sup>6</sup> The `%` operator is also used for string formatting; the same precedence applies.



---

## 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
           | type_stmt
```

### 7.1 Expression statements

Expression statements are used (mostly interactively) to compute and write a value, or (usually) to call a procedure (a function that returns no meaningful result; in Python, procedures return the value `None`). Other uses of expression statements are allowed and occasionally useful. The syntax for an expression statement is:

```
expression_stmt: starred_expression
```

An expression statement evaluates the expression list (which may be a single expression).

In interactive mode, if the value is not `None`, it is converted to a string using the built-in `repr()` function and the resulting string is written to standard output on a line by itself (except if the result is `None`, so that procedure calls do not cause any output.)

## 7.2 Assignment statements

Assignment statements are used to (re)bind names to values and to modify attributes or items of mutable objects:

```
assignment_stmt: (target_list "=") + (starred_expression | yield_expression)
target_list:      target ("," target) * [","]
target:           identifier
                  | "(" [target_list] ")"
                  | "[" [target_list] "]"
                  | attributeref
                  | subscription
                  | slicing
                  | "*" target
```

(See section [Primaries](#) for the syntax definitions for *attributeref*, *subscription*, and *slicing*.)

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section [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/value pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

For user-defined objects, the `__setitem__()` method is called with appropriate arguments.

- If the target is a slicing: The primary expression in the reference is evaluated. It should yield a mutable sequence object (such as a list). The assigned object should be a sequence object of the same type. Next, the lower and upper bound expressions are evaluated, insofar they are present; defaults are zero and the sequence's length. The bounds should evaluate to integers. If either bound is negative, the sequence's length is added to it. The resulting bounds are clipped to lie between zero and the sequence's length, inclusive. Finally, the sequence object is asked to replace the slice with the items of the assigned sequence. The length of the slice may be different from the length of the assigned sequence, thus changing the length of the target sequence, if the target sequence allows it.

**Λεπτομέρεια υλοποίησης CPython:** 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 [Primitives](#) for the syntax definitions of the last three symbols.)

An augmented assignment evaluates the target (which, unlike normal assignment statements, cannot be an unpacking) and the expression list, performs the binary operation specific to the type of assignment on the two operands, and assigns the result to the original target. The target is only evaluated once.

An augmented assignment statement like `x += 1` can be rewritten as `x = x + 1` to achieve a similar, but not exactly equal effect. In the augmented version, `x` is only evaluated once. Also, when possible, the actual operation is performed *in-place*, meaning that rather than creating a new object and assigning that to the target, the old object is modified instead.

Unlike normal assignments, augmented assignments evaluate the left-hand side *before* evaluating the right-hand side. For example, `a[i] += f(x)` first looks-up `a[i]`, then it evaluates `f(x)` and performs the addition, and lastly, it writes the result back to `a[i]`.

With the exception of assigning to tuples and multiple targets in a single statement, the assignment done by augmented assignment statements is handled the same way as normal assignments. Similarly, with the exception of the possible *in-place* behavior, the binary operation performed by augmented assignment is the same as the normal binary operations.

For targets which are attribute references, the same *caveat about class and instance attributes* applies as for regular assignments.

## 7.2.2 Annotated assignment statements

*Annotation* assignment is the combination, in a single statement, of a variable or attribute annotation and an optional assignment statement:

```
annotated_assignment_stmt: augtarget ":" expression
                           [ "=" (starred_expression | yield_expression) ]
```

The difference from normal *Assignment statements* is that only a single target is allowed.

The assignment target is considered «simple» if it consists of a single name that is not enclosed in parentheses. For simple assignment targets, if in class or module scope, the annotations are gathered in a lazily evaluated *annotation scope*. The annotations can be evaluated using the `__annotations__` attribute of a class or module, or using the facilities in the `annotationlib` module.

If the assignment target is not simple (an attribute, subscript node, or parenthesized name), the annotation is never evaluated.

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 as if there was no annotation present. 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 the same expressions in the right hand side as regular assignments. Previously, some expressions (like un-parenthesized tuple expressions) caused a syntax error.

Άλλαξε στην έκδοση 3.14: Annotations are now lazily evaluated in a separate *annotation scope*. If the assignment target is not simple, annotations are never evaluated.

## 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).

Αλλάξε 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. You can create an exception and set your own traceback in one step using the `with_traceback()` exception method (which returns the same exception instance, with its traceback set to its argument), like so:

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

The `from` clause is used for exception chaining: if given, the second *expression* must be another exception class or instance. If the second expression is an exception instance, it will be attached to the raised exception as the `__cause__` attribute (which is writable). If the expression is an exception class, the class will be instantiated and the resulting exception instance will be attached to the raised exception as the `__cause__` attribute. If the raised exception is not handled, both exceptions will be printed:

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

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

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

A similar mechanism works implicitly if a new exception is raised when an exception is already being handled. An exception may be handled when an *except* or *finally* clause, or a *with* statement, is used. The previous exception is then attached as the new exception's `__context__` attribute:

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

During handling of the above exception, another exception occurred:

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

Exception chaining can be explicitly suppressed by specifying `None` in the `from` clause:

```
>>> try:
...     print(1 / 0)
... except:
...     raise RuntimeError("Something bad happened") from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: Something bad happened
```

Additional information on exceptions can be found in section *Εξαίρεσεις*, 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`.

Added the `__suppress_context__` attribute to suppress automatic display of the exception context.

Άλλαξε στην έκδοση 3.11: If the traceback of the active exception is modified in an `except` clause, a subsequent `raise` statement re-raises the exception with the modified traceback. Previously, the exception was re-raised with the traceback it had when it was caught.

## 7.9 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 causes the listed identifiers to be interpreted as globals. It would be impossible to assign to a global variable without `global`, although free variables may refer to globals without being declared `global`.

The `global` statement applies to the entire scope of a function or class body. A `SyntaxError` is raised if a variable is used or assigned to prior to its global declaration in the scope.

**Programmer's note:** `global` is a directive to the parser. It applies only to code parsed at the same time as the `global` statement. In particular, a `global` statement contained in a string or code object supplied to the built-in `exec()` function does not affect the code block *containing* the function call, and code contained in such a string is unaffected by `global` statements in the code containing the function call. The same applies to the `eval()` and `compile()` functions.

## 7.13 The `nonlocal` statement

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

When the definition of a function or class is nested (enclosed) within the definitions of other functions, its `nonlocal` scopes are the local scopes of the enclosing functions. The `nonlocal` statement causes the listed identifiers to refer to names previously bound in `nonlocal` scopes. It allows encapsulated code to rebind such `nonlocal` identifiers. If a name is bound in more than one `nonlocal` scope, the nearest binding is used. If a name is not bound in any `nonlocal` scope, or if there is no `nonlocal` scope, a `SyntaxError` is raised.

The `nonlocal` statement applies to the entire scope of a function or class body. A `SyntaxError` is raised if a variable is used or assigned to prior to its `nonlocal` declaration in the scope.

➡ Δείτε επίσης

**PEP 3104 - Access to Names in Outer Scopes**

The specification for the `nonlocal` statement.

**Programmer's note:** `nonlocal` is a directive to the parser and applies only to code parsed along with it. See the note for the `global` statement.

## 7.14 The `type` statement

```
type_stmt: 'type' identifier [type_params] "=" expression
```

The `type` statement declares a type alias, which is an instance of `typing.TypeAliasType`.

For example, the following statement creates a type alias:

```
type Point = tuple[float, float]
```

This code is roughly equivalent to:

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

`annotation-def` indicates an *annotation scope*, which behaves mostly like a function, but with several small differences.

The value of the type alias is evaluated in the annotation scope. It is not evaluated when the type alias is created, but only when the value is accessed through the type alias's `__value__` attribute (see *Καθυστέρημένη εκτίμηση*). This allows the type alias to refer to names that are not yet defined.

Type aliases may be made generic by adding a *type parameter list* after the name. See *Generic type aliases* for more. `type` is a *soft keyword*.

Added in version 3.12.

 Δείτε επίσης

### PEP 695 - Type Parameter Syntax

Introduced the `type` statement and syntax for generic classes and functions.

---

## 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
               | match_stmt
               | funcdef
               | classdef
               | async_with_stmt
               | async_for_stmt
               | async_funcdef
suite:         stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement:    stmt_list NEWLINE | compound_stmt
```

```
stmt_list:      simple_stmt (";" simple_stmt) * [";"]
```

Note that statements always end in a NEWLINE possibly followed by a DEDENT. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the “dangling *else*” problem is solved in Python by requiring nested *if* statements to be indented).

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.

## 8.1 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" starred_expression_list ":" suite
          ["else" ":" suite]
```

The *starred\_expression\_list* expression is evaluated once; it should yield an *iterable* object. An *iterator* is created for that iterable. The first item provided by the iterator is then assigned to the target list using the standard rules for assignments (see *Assignment statements*), and the suite is executed. This repeats for each item provided by the iterator. When the iterator is exhausted, the suite in the *else* clause, if present, is executed, and the loop terminates.

A *break* statement executed in the first suite terminates the loop without executing the *else* clause’s suite. A *continue* statement executed in the first suite skips the rest of the suite and continues with the next item, or with the *else* clause if there is no next item.

The for-loop makes assignments to the variables in the target list. This overwrites all previous assignments to those variables including those made in the suite of the for-loop:

```
for i in range(10):
    print(i)
    i = 5                                # this will not affect the for-loop
                                        # because i will be overwritten with the next
                                        # index in the range
```

Names in the target list are not deleted when the loop is finished, but if the sequence is empty, they will not have been assigned to at all by the loop. Hint: the built-in type `range()` represents immutable arithmetic sequences of integers. For instance, iterating `range(3)` successively yields 0, 1, and then 2.

Άλλαξε στην έκδοση 3.11: Starred elements are now allowed in the expression list.

## 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 | try3_stmt
try1_stmt: "try" ":" suite
          ("except" [expression ["as" identifier]] ":" suite)+
          ["else" ":" suite]
          ["finally" ":" suite]
try2_stmt: "try" ":" suite
          ("except" "*" expression ["as" identifier] ":" suite)+
          ["else" ":" suite]
          ["finally" ":" suite]
try3_stmt: "try" ":" suite
          "finally" ":" suite
```

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

Άλλαξε στην έκδοση 3.14: Support for optionally dropping grouping parentheses when using multiple exception types. See [PEP 758](#).

### 8.4.1 `except` clause

The `except` clause(s) specify one or more exception handlers. When no exception occurs in the `try` clause, no exception handler is executed. When an exception occurs in the `try` suite, a search for an exception handler is started. This search inspects the `except` clauses in turn until one is found that matches the exception. An expression-less `except` clause, if present, must be last; it matches any exception.

For an `except` clause with an expression, the expression must evaluate to an exception type or a tuple of exception types. Parentheses can be dropped if multiple exception types are provided and the `as` clause is not used. The raised exception matches an `except` clause whose expression evaluates to the class or a *non-virtual base class* of the exception object, or to a tuple that contains such a class.

If no `except` clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack.<sup>1</sup>

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

<sup>1</sup> 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.

```

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, the exception is stored in the `sys` module, where it can be accessed from within the body of the `except` clause by calling `sys.exception()`. When leaving an exception handler, the exception stored in the `sys` module is reset to its previous value:

```

>>> print(sys.exception())
None
>>> try:
...     raise TypeError
... except:
...     print(repr(sys.exception()))
...     try:
...         raise ValueError
...     except:
...         print(repr(sys.exception()))
...         print(repr(sys.exception()))
...
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None

```

### 8.4.2 `except*` clause

The `except*` clause(s) are used for handling `ExceptionGroups`. The exception type for matching is interpreted as in the case of `except`, but in the case of exception groups we can have partial matches when the type matches some of the exceptions in the group. This means that multiple `except*` clauses can execute, each handling part of the exception group. Each clause executes at most once and handles an exception group of all matching exceptions. Each exception in the group is handled by at most one `except*` clause, the first that matches it.

```

>>> try:
...     raise ExceptionGroup("eg",
...                           [ValueError(1), TypeError(2), OSError(3), OSError(4)])
... except* TypeError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
... except* OSError as e:
...     print(f'caught {type(e)} with nested {e.exceptions}')
...
caught <class 'ExceptionGroup'> with nested (TypeError(2),)
caught <class 'ExceptionGroup'> with nested (OSError(3), OSError(4))
+ Exception Group Traceback (most recent call last):
+ |   File "<stdin>", line 2, in <module>
+ | ExceptionGroup: eg
+ +-+----- 1 -----+
+ | ValueError: 1
+ +-----+

```

Any remaining exceptions that were not handled by any `except*` clause are re-raised at the end, along with all



exceptions that were raised from within the `except *` clauses. If this list contains more than one exception to reraise, they are combined into an exception group.

If the raised exception is not an exception group and its type matches one of the `except *` clauses, it is caught and wrapped by an exception group with an empty message string.

```
>>> try:
...     raise BlockingIOError
... except* BlockingIOError as e:
...     print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

An `except *` clause must have a matching expression; it cannot be `except *:`. Furthermore, this expression cannot contain exception group types, because that would have ambiguous semantics.

It is not possible to mix `except` and `except *` in the same `try`. `break`, `continue` and `return` cannot appear in an `except *` clause.

### 8.4.3 `else` clause

The optional `else` clause is executed if the control flow leaves the `try` suite, no exception was raised, and no `return`, `continue`, or `break` statement was executed. Exceptions in the `else` clause are not handled by the preceding `except` clauses.

### 8.4.4 `finally` clause

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. For example, this function returns 42.

```
def f():
    try:
        1/0
    finally:
        return 42
```

The exception information is not available to the program during execution of the `finally` clause.

When a `return`, `break` or `continue` statement is executed in the `try` suite of a `try...finally` statement, the `finally` clause is also executed “on the way out.”

The return value of a function is determined by the last `return` statement executed. Since the `finally` clause always executes, a `return` statement executed in the `finally` clause will always be the last one executed. The following function returns “finally”.

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

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.

Άλλαξε στην έκδοση 3.14: The compiler emits a `SyntaxWarning` when a `return`, `break` or `continue` appears in a `finally` block (see [PEP 765](#)).

## 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_stmt_contents "," "?" ")" | with_stmt_contents ) ":" suite
with_stmt_contents: with_item ( "," with_item ) *
with_item:      expression ["as" target]
```

The execution of the `with` statement with one «item» proceeds as follows:

1. The context expression (the expression given in the `with_item`) is evaluated to obtain a context manager.
2. The context manager's `__enter__()` is loaded for later use.
3. The context manager's `__exit__()` is loaded for later use.
4. The context manager's `__enter__()` method is invoked.
5. If a target was included in the `with` statement, the return value from `__enter__()` is assigned to it.

### 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 7 below.

6. The suite is executed.
7. The context manager's `__exit__()` method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to `__exit__()`. Otherwise, three `None` arguments are supplied.

If the suite was exited due to an exception, and the return value from the `__exit__()` method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the `with` statement.

If the suite was exited for any reason other than an exception, the return value from `__exit__()` is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

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:
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
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
```

You can also write multi-item context managers in multiple lines if the items are surrounded by parentheses. For example:

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

Άλλαξε στην έκδοση 3.1: Support for multiple context expressions.

Άλλαξε στην έκδοση 3.10: Support for using grouping parentheses to break the statement in multiple lines.

### ➡ Δείτε επίσης

#### PEP 343 - The «with» statement

The specification, background, and examples for the Python *with* statement.

## 8.6 The match statement

Added in version 3.10.

The match statement is used for pattern matching. Syntax:

```
match_stmt:  'match' subject_expr ":" NEWLINE INDENT case_block+ DEDENT
subject_expr: star_named_expression ", " star_named_expressions?
              | named_expression
case_block:  'case' patterns [guard] ":" block
```

### ℹ Σημείωση

This section uses single quotes to denote *soft keywords*.

Pattern matching takes a pattern as input (following *case*) and a subject value (following *match*). The pattern (which may contain subpatterns) is matched against the subject value. The outcomes are:

- A match success or failure (also termed a pattern success or failure).
- Possible binding of matched values to a name. The prerequisites for this are further discussed below.

The *match* and *case* keywords are *soft keywords*.

### ➡ Δείτε επίσης

- [PEP 634](#) – Structural Pattern Matching: Specification
- [PEP 636](#) – Structural Pattern Matching: Tutorial

## 8.6.1 Overview

Here's an overview of the logical flow of a match statement:

1. The subject expression `subject_expr` is evaluated and a resulting subject value obtained. If the subject expression contains a comma, a tuple is constructed using the standard rules.
2. Each pattern in a `case_block` is attempted to match with the subject value. The specific rules for success or failure are described below. The match attempt can also bind some or all of the standalone names within the pattern. The precise pattern binding rules vary per pattern type and are specified below. **Name bindings made during a successful pattern match outlive the executed block and can be used after the match statement.**

### ℹ Σημείωση

During failed pattern matches, some subpatterns may succeed. Do not rely on bindings being made for a failed match. Conversely, do not rely on variables remaining unchanged after a failed match. The exact behavior is dependent on implementation and may vary. This is an intentional decision made to allow different implementations to add optimizations.

3. If the pattern succeeds, the corresponding guard (if present) is evaluated. In this case all name bindings are guaranteed to have happened.
  - If the guard evaluates as true or is missing, the block inside `case_block` is executed.
  - Otherwise, the next `case_block` is attempted as described above.
  - If there are no further case blocks, the match statement is completed.

### ℹ Σημείωση

Users should generally never rely on a pattern being evaluated. Depending on implementation, the interpreter may cache values or use other optimizations which skip repeated evaluations.

A sample match statement:

```
>>> flag = False
>>> match (100, 200):
...     case (100, 300): # Mismatch: 200 != 300
...         print('Case 1')
...     case (100, 200) if flag: # Successful match, but guard fails
...         print('Case 2')
...     case (100, y): # Matches and binds y to 200
...         print(f'Case 3, y: {y}')
...     case _: # Pattern not attempted
...         print('Case 4, I match anything!')
...
Case 3, y: 200
```

In this case, `if flag` is a guard. Read more about that in the next section.

### 8.6.2 Guards

```
guard: "if" named_expression
```

A guard (which is part of the `case`) must succeed for code inside the `case` block to execute. It takes the form: *if* followed by an expression.

The logical flow of a `case` block with a guard follows:

1. Check that the pattern in the `case` block succeeded. If the pattern failed, the guard is not evaluated and the next `case` block is checked.
2. If the pattern succeeded, evaluate the guard.
  - If the guard condition evaluates as true, the case block is selected.
  - If the guard condition evaluates as false, the case block is not selected.
  - If the guard raises an exception during evaluation, the exception bubbles up.

Guards are allowed to have side effects as they are expressions. Guard evaluation must proceed from the first to the last case block, one at a time, skipping case blocks whose pattern(s) don't all succeed. (I.e., guard evaluation must happen in order.) Guard evaluation must stop once a case block is selected.

### 8.6.3 Irrefutable Case Blocks

An irrefutable case block is a match-all case block. A match statement may have at most one irrefutable case block, and it must be last.

A case block is considered irrefutable if it has no guard and its pattern is irrefutable. A pattern is considered irrefutable if we can prove from its syntax alone that it will always succeed. Only the following patterns are irrefutable:

- *AS Patterns* whose left-hand side is irrefutable
- *OR Patterns* containing at least one irrefutable pattern
- *Capture Patterns*
- *Wildcard Patterns*
- parenthesized irrefutable patterns

### 8.6.4 Patterns

#### Σημείωση

This section uses grammar notations beyond standard EBNF:

- the notation `SEP . RULE+` is shorthand for `RULE (SEP RULE) *`
- the notation `!RULE` is shorthand for a negative lookahead assertion

The top-level syntax for patterns is:

```
patterns:      open_sequence_pattern | pattern
pattern:      as_pattern | or_pattern
closed_pattern: | literal_pattern
               | capture_pattern
               | wildcard_pattern
               | value_pattern
               | group_pattern
               | sequence_pattern
               | mapping_pattern
               | class_pattern
```

The descriptions below will include a description «in simple terms» of what a pattern does for illustration purposes (credits to Raymond Hettinger for a document that inspired most of the descriptions). Note that these descriptions are purely for illustration purposes and **may not** reflect the underlying implementation. Furthermore, they do not cover all valid forms.

## OR Patterns

An OR pattern is two or more patterns separated by vertical bars `|`. Syntax:

```
or_pattern: "|" closed_pattern+
```

Only the final subpattern may be *irrefutable*, and each subpattern must bind the same set of names to avoid ambiguity.

An OR pattern matches each of its subpatterns in turn to the subject value, until one succeeds. The OR pattern is then considered successful. Otherwise, if none of the subpatterns succeed, the OR pattern fails.

In simple terms, `P1 | P2 | ...` will try to match `P1`, if it fails it will try to match `P2`, succeeding immediately if any succeeds, failing otherwise.

## AS Patterns

An AS pattern matches an OR pattern on the left of the *as* keyword against a subject. Syntax:

```
as_pattern: or_pattern "as" capture_pattern
```

If the OR pattern fails, the AS pattern fails. Otherwise, the AS pattern binds the subject to the name on the right of the *as* keyword and succeeds. *capture\_pattern* cannot be a `_`.

In simple terms `P as NAME` will match with `P`, and on success it will set `NAME = <subject>`.

## Literal Patterns

A literal pattern corresponds to most *literals* in Python. Syntax:

```
literal_pattern: signed_number
                | signed_number "+" NUMBER
                | signed_number "-" NUMBER
                | strings
                | "None"
                | "True"
                | "False"
signed_number:  ["-"] NUMBER
```

The rule *strings* and the token `NUMBER` are defined in the *standard Python grammar*. Triple-quoted strings are supported. Raw strings and byte strings are supported. *f-strings* and *t-strings* are not supported.

The forms `signed_number '+' NUMBER` and `signed_number '-' NUMBER` are for expressing *complex numbers*; they require a real number on the left and an imaginary number on the right. E.g. `3 + 4j`.

In simple terms, `LITERAL` will succeed only if `<subject> == LITERAL`. For the singletons `None`, `True` and `False`, the *is* operator is used.

## Capture Patterns

A capture pattern binds the subject value to a name. Syntax:

```
capture_pattern: !'_' NAME
```

A single underscore `_` is not a capture pattern (this is what `! '_'` expresses). It is instead treated as a *wildcard\_pattern*.

In a given pattern, a given name can only be bound once. E.g. `case x, x: ...` is invalid while `case [x] | x: ...` is allowed.

Capture patterns always succeed. The binding follows scoping rules established by the assignment expression operator in [PEP 572](#); the name becomes a local variable in the closest containing function scope unless there's an applicable *global* or *nonlocal* statement.

In simple terms `NAME` will always succeed and it will set `NAME = <subject>`.

## Wildcard Patterns

A wildcard pattern always succeeds (matches anything) and binds no name. Syntax:

```
wildcard_pattern: '_'
```

`_` is a *soft keyword* within any pattern, but only within patterns. It is an identifier, as usual, even within `match` subject expressions, guards, and case blocks.

In simple terms, `_` will always succeed.

## Value Patterns

A value pattern represents a named value in Python. Syntax:

```
value_pattern: attr
attr:         name_or_attr "." NAME
name_or_attr: attr | NAME
```

The dotted name in the pattern is looked up using standard Python *name resolution rules*. The pattern succeeds if the value found compares equal to the subject value (using the `==` equality operator).

In simple terms `NAME1.NAME2` will succeed only if `<subject> == NAME1.NAME2`

### Σημείωση

If the same value occurs multiple times in the same match statement, the interpreter may cache the first value found and reuse it rather than repeat the same lookup. This cache is strictly tied to a given execution of a given match statement.

## Group Patterns

A group pattern allows users to add parentheses around patterns to emphasize the intended grouping. Otherwise, it has no additional syntax. Syntax:

```
group_pattern: "(" pattern ")"
```

In simple terms `(P)` has the same effect as `P`.

## Sequence Patterns

A sequence pattern contains several subpatterns to be matched against sequence elements. The syntax is similar to the unpacking of a list or tuple.

```
sequence_pattern: "[" [maybe_sequence_pattern] "]"
                  | "(" [open_sequence_pattern] ")"
open_sequence_pattern: maybe_star_pattern "," [maybe_sequence_pattern]
maybe_sequence_pattern: "," . maybe_star_pattern+ "," "?"
maybe_star_pattern: star_pattern | pattern
star_pattern: "*" (capture_pattern | wildcard_pattern)
```

There is no difference if parentheses or square brackets are used for sequence patterns (i.e. `(...)` vs `[...]`).

**Σημείωση**

A single pattern enclosed in parentheses without a trailing comma (e.g. `(3 | 4)`) is a *group pattern*. While a single pattern enclosed in square brackets (e.g. `[3 | 4]`) is still a sequence pattern.

At most one star subpattern may be in a sequence pattern. The star subpattern may occur in any position. If no star subpattern is present, the sequence pattern is a fixed-length sequence pattern; otherwise it is a variable-length sequence pattern.

The following is the logical flow for matching a sequence pattern against a subject value:

1. If the subject value is not a sequence<sup>2</sup>, the sequence pattern fails.
2. If the subject value is an instance of `str`, `bytes` or `bytearray` the sequence pattern fails.
3. The subsequent steps depend on whether the sequence pattern is fixed or variable-length.

If the sequence pattern is fixed-length:

1. If the length of the subject sequence is not equal to the number of subpatterns, the sequence pattern fails
2. Subpatterns in the sequence pattern are matched to their corresponding items in the subject sequence from left to right. Matching stops as soon as a subpattern fails. If all subpatterns succeed in matching their corresponding item, the sequence pattern succeeds.

Otherwise, if the sequence pattern is variable-length:

1. If the length of the subject sequence is less than the number of non-star subpatterns, the sequence pattern fails.
2. The leading non-star subpatterns are matched to their corresponding items as for fixed-length sequences.
3. If the previous step succeeds, the star subpattern matches a list formed of the remaining subject items, excluding the remaining items corresponding to non-star subpatterns following the star subpattern.
4. Remaining non-star subpatterns are matched to their corresponding subject items, as for a fixed-length sequence.

**Σημείωση**

The length of the subject sequence is obtained via `len()` (i.e. via the `__len__()` protocol). This length may be cached by the interpreter in a similar manner as *value patterns*.

In simple terms `[P1, P2, P3, ..., P<N>]` matches only if all the following happens:

- `check <subject>` is a sequence
- `len(subject) == <N>`

<sup>2</sup> In pattern matching, a sequence is defined as one of the following:

- a class that inherits from `collections.abc.Sequence`
- a Python class that has been registered as `collections.abc.Sequence`
- a builtin class that has its (CPython) `Py_TPFLAGS_SEQUENCE` bit set
- a class that inherits from any of the above

The following standard library classes are sequences:

- `array.array`
- `collections.deque`
- `list`
- `memoryview`
- `range`
- `tuple`

**Σημείωση**

Subject values of type `str`, `bytes`, and `bytearray` do not match sequence patterns.



- P1 matches <subject>[0] (note that this match can also bind names)
- P2 matches <subject>[1] (note that this match can also bind names)
- ... and so on for the corresponding pattern/element.

## Mapping Patterns

A mapping pattern contains one or more key-value patterns. The syntax is similar to the construction of a dictionary. Syntax:

```
mapping_pattern:      "{" [items_pattern] "}"
items_pattern:        "," .key_value_pattern+ "," "?"
key_value_pattern:    (literal_pattern | value_pattern) ":" pattern
                      | double_star_pattern
double_star_pattern:  "***" capture_pattern
```

At most one double star pattern may be in a mapping pattern. The double star pattern must be the last subpattern in the mapping pattern.

Duplicate keys in mapping patterns are disallowed. Duplicate literal keys will raise a `SyntaxError`. Two keys that otherwise have the same value will raise a `ValueError` at runtime.

The following is the logical flow for matching a mapping pattern against a subject value:

1. If the subject value is not a mapping<sup>3</sup>, the mapping pattern fails.
2. If every key given in the mapping pattern is present in the subject mapping, and the pattern for each key matches the corresponding item of the subject mapping, the mapping pattern succeeds.
3. If duplicate keys are detected in the mapping pattern, the pattern is considered invalid. A `SyntaxError` is raised for duplicate literal values; or a `ValueError` for named keys of the same value.

### Σημείωση

Key-value pairs are matched using the two-argument form of the mapping subject's `get()` method. Matched key-value pairs must already be present in the mapping, and not created on-the-fly via `__missing__()` or `__getitem__()`.

In simple terms `{KEY1: P1, KEY2: P2, ... }` matches only if all the following happens:

- check <subject> is a mapping
- KEY1 in <subject>
- P1 matches <subject>[KEY1]
- ... and so on for the corresponding KEY/pattern pair.

## Class Patterns

A class pattern represents a class and its positional and keyword arguments (if any). Syntax:

```
class_pattern:        name_or_attr "(" [pattern_arguments "," "?" ] ")"
pattern_arguments:    positional_patterns ["," keyword_patterns]
                      | keyword_patterns
positional_patterns:  "," .pattern+
keyword_patterns:     "," .keyword_pattern+
```

<sup>3</sup> In pattern matching, a mapping is defined as one of the following:

- a class that inherits from `collections.abc.Mapping`
- a Python class that has been registered as `collections.abc.Mapping`
- a builtin class that has its (CPython) `Py_TPFLAGS_MAPPING` bit set
- a class that inherits from any of the above

The standard library classes `dict` and `types.MappingProxyType` are mappings.

`keyword_pattern:`        `NAME` `"="` *pattern*

The same keyword should not be repeated in class patterns.

The following is the logical flow for matching a class pattern against a subject value:

1. If `name_or_attr` is not an instance of the builtin `type`, raise `TypeError`.
2. If the subject value is not an instance of `name_or_attr` (tested via `isinstance()`), the class pattern fails.
3. If no pattern arguments are present, the pattern succeeds. Otherwise, the subsequent steps depend on whether keyword or positional argument patterns are present.

For a number of built-in types (specified below), a single positional subpattern is accepted which will match the entire subject; for these types keyword patterns also work as for other types.

If only keyword patterns are present, they are processed as follows, one by one:

I. The keyword is looked up as an attribute on the subject.

- If this raises an exception other than `AttributeError`, the exception bubbles up.
- If this raises `AttributeError`, the class pattern has failed.
- Else, the subpattern associated with the keyword pattern is matched against the subject's attribute value. If this fails, the class pattern fails; if this succeeds, the match proceeds to the next keyword.

II. If all keyword patterns succeed, the class pattern succeeds.

If any positional patterns are present, they are converted to keyword patterns using the `__match_args__` attribute on the class `name_or_attr` before matching:

I. The equivalent of `getattr(cls, "__match_args__", ())` is called.

- If this raises an exception, the exception bubbles up.
- If the returned value is not a tuple, the conversion fails and `TypeError` is raised.
- If there are more positional patterns than `len(cls.__match_args__)`, `TypeError` is raised.
- Otherwise, positional pattern `i` is converted to a keyword pattern using `__match_args__[i]` as the keyword. `__match_args__[i]` must be a string; if not `TypeError` is raised.
- If there are duplicate keywords, `TypeError` is raised.

 **Δείτε επίσης**

*Customizing positional arguments in class pattern matching*

**II. Once all positional patterns have been converted to keyword patterns,**  
the match proceeds as if there were only keyword patterns.

For the following built-in types the handling of positional subpatterns is different:

- `bool`
- `bytearray`
- `bytes`
- `dict`
- `float`
- `frozenset`
- `int`

- list
- set
- str
- tuple

These classes accept a single positional argument, and the pattern there is matched against the whole object rather than an attribute. For example `int(0|1)` matches the value 0, but not the value 0.0.

In simple terms `CLS(P1, attr=P2)` matches only if the following happens:

- `isinstance(<subject>, CLS)`
- convert P1 to a keyword pattern using `CLS.__match_args__`
- For each keyword argument `attr=P2`:
  - `hasattr(<subject>, "attr")`
  - P2 matches `<subject>.attr`
- ... and so on for the corresponding keyword argument/pattern pair.

#### ➡ Δείτε επίσης

- [PEP 634](#) – Structural Pattern Matching: Specification
- [PEP 636](#) – Structural Pattern Matching: Tutorial

## 8.7 Function definitions

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

```
funcdef:          [decorators] "def" funcname [type_params] "(" [parameter_list] ")" "
                  ["->" expression] ":" suite
decorators:       decorator+
decorator:        "@" assignment_expression NEWLINE
parameter_list:   defparameter ("," defparameter)* "," "/" ["," [parameter_list_no_posonly
                  | parameter_list_no_posonly
parameter_list_no_posonly: defparameter ("," defparameter)* ["," [parameter_list_starargs
                  | parameter_list_starargs
parameter_list_starargs:  "*" [star_parameter] ("," defparameter)* ["," [parameter_star_kwarg
                  | "*" ("," defparameter)+ ["," [parameter_star_kwarg]]
                  | parameter_star_kwarg
parameter_star_kwarg:    "***" parameter [","]
parameter:         identifier [":" expression]
star_parameter:     identifier [":" ["*"] expression]
defparameter:       parameter ["=" expression]
funcname:          identifier
```

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called.<sup>4</sup>

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

<sup>4</sup> 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*.

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.

A list of *type parameters* may be given in square brackets between the function's name and the opening parenthesis for its parameter list. This indicates to static type checkers that the function is generic. At runtime, the type parameters can be retrieved from the function's `__type_params__` attribute. See [Generic functions](#) for more.

Άλλαξε στην έκδοση 3.12: Type parameter lists are new in Python 3.12.

When one or more *parameters* have the form *parameter = expression*, the function is said to have «default parameter values.» For a parameter with a default value, the corresponding *argument* may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters up until the «\*» must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

**Default parameter values are evaluated from left to right when the function definition is executed.** This means that the expression is evaluated once, when the function is defined, and that the same «pre-computed» value is used for each call. This is especially important to understand when a default parameter value is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default parameter value is in effect modified. This is generally not what was intended. A way around this is to use `None` as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section [Calls](#). A function call always assigns values to all parameters mentioned in the parameter list, either from positional arguments, from keyword arguments, or from default values. If the form «*identifier*» is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form «*\*\*identifier*» is present, it is initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type. Parameters after «\*» or «*identifier*» are keyword-only parameters and may only be passed by keyword arguments. Parameters before «/» are positional-only parameters and may only be passed by positional arguments.

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*. (As a special case, parameters of the form *\*identifier* may have an annotation «`: *expression`».) Functions may have «return» annotation of the form «`-> expression`» after the parameter list. These annotations can be any valid Python expression. The presence of annotations does not change the semantics of a function. See [Annotations](#) for more information on annotations.

Άλλαξε στην έκδοση 3.11: Parameters of the form «*\*identifier*» may have an annotation «`: *expression`». See [PEP 646](#).

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda expressions, described in section [Lambdas](#). Note that the lambda expression is merely a shorthand for a simplified function definition; a function defined in a «def» statement can be passed around or assigned to another name just like a function defined by a lambda expression. The «def» form is actually more powerful since it allows the execution of multiple statements and annotations.

**Programmer’s note:** Functions are first-class objects. A «def» statement executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section [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.

#### PEP 318 - Decorators for Functions and Methods

Function and method decorators were introduced. Class decorators were introduced in [PEP 3129](#).

## 8.8 Class definitions

A class definition defines a class object (see section [The standard type hierarchy](#)):

```
classdef:    [decorators] "class" classname [type_params] [inheritance] ":" suite
inheritance: "(" [argument_list] ")"
classname:  identifier
```

A class definition is an executable statement. The inheritance list usually gives a list of base classes (see [Metaclasses](#) for more advanced uses), so each item in the list should evaluate to a class object which allows subclassing. Classes without an inheritance list inherit, by default, from the base class `object`; hence,

```
class Foo:
    pass
```

is equivalent to

```
class Foo(object):
    pass
```

The class’s suite is then executed in a new execution frame (see [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.<sup>5</sup> 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](#).

<sup>5</sup> 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*.

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.

A list of *type parameters* may be given in square brackets immediately after the class's name. This indicates to static type checkers that the class is generic. At runtime, the type parameters can be retrieved from the class's `__type_params__` attribute. See *Generic classes* for more.

Άλλαξε στην έκδοση 3.12: Type parameter lists are new in Python 3.12.

**Programmer's note:** Variables defined in the class definition are class attributes; they are shared by instances. Instance attributes can be set in a method with `self.name = value`. Both class and instance attributes are accessible through the notation `«self.name»`, and an instance attribute hides a class attribute with the same name when accessed in this way. Class attributes can be used as defaults for instance attributes, but using mutable values there can lead to unexpected results. *Descriptors* can be used to create instance variables with different implementation details.

### ➡ 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](#).

## 8.9 Coroutines

Added in version 3.5.

### 8.9.1 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*). *await* expressions, *async for* and *async with* can only be used in the body of a coroutine function.

Functions defined with `async def` syntax are always coroutine functions, even if they do not contain `await` or `async` keywords.

It is a `SyntaxError` to use a `yield from` expression inside the body of a coroutine function.

An example of a coroutine function:

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

Άλλαξε στην έκδοση 3.7: `await` and `async` are now keywords; previously they were only treated as such inside the body of a coroutine function.

## 8.9.2 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.

## 8.9.3 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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

    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.

## 8.10 Type parameter lists

Added in version 3.12.

Άλλαξε στην έκδοση 3.13: Support for default values was added (see [PEP 696](#)).

```

type_params: "[" type_param ("," type_param)* "]"
type_param:  typevar | typevartuple | paramspec
typevar:     identifier (":" expression)? ("=" expression)?
typevartuple: "*" identifier ("=" expression)?
paramspec:   "*" identifier ("=" expression)?

```

*Functions* (including *coroutines*), *classes* and *type aliases* may contain a type parameter list:

```

def max[T](args: list[T]) -> T:
    ...

async def amax[T](args: list[T]) -> T:
    ...

class Bag[T]:
    def __iter__(self) -> Iterator[T]:
        ...

    def add(self, arg: T) -> None:
        ...

type ListOrSet[T] = list[T] | set[T]

```

Semantically, this indicates that the function, class, or type alias is generic over a type variable. This information is primarily used by static type checkers, and at runtime, generic objects behave much like their non-generic counterparts.

Type parameters are declared in square brackets (`[]`) immediately after the name of the function, class, or type alias. The type parameters are accessible within the scope of the generic object, but not elsewhere. Thus, after a declaration `def func[T]():` `pass`, the name `T` is not available in the module scope. Below, the semantics of generic objects are described with more precision. The scope of type parameters is modeled with a special function (technically, an *annotation scope*) that wraps the creation of the generic object.

Generic functions, classes, and type aliases have a `__type_params__` attribute listing their type parameters.



Type parameters come in three kinds:

- `typing.TypeVar`, introduced by a plain name (e.g., `T`). Semantically, this represents a single type to a type checker.
- `typing.TypeVarTuple`, introduced by a name prefixed with a single asterisk (e.g., `*Ts`). Semantically, this stands for a tuple of any number of types.
- `typing.ParamSpec`, introduced by a name prefixed with two asterisks (e.g., `**P`). Semantically, this stands for the parameters of a callable.

`typing.TypeVar` declarations can define *bounds* and *constraints* with a colon (`:`) followed by an expression. A single expression after the colon indicates a bound (e.g. `T: int`). Semantically, this means that the `typing.TypeVar` can only represent types that are a subtype of this bound. A parenthesized tuple of expressions after the colon indicates a set of constraints (e.g. `T: (str, bytes)`). Each member of the tuple should be a type (again, this is not enforced at runtime). Constrained type variables can only take on one of the types in the list of constraints.

For `typing.TypeVars` declared using the type parameter list syntax, the bound and constraints are not evaluated when the generic object is created, but only when the value is explicitly accessed through the attributes `__bound__` and `__constraints__`. To accomplish this, the bounds or constraints are evaluated in a separate *annotation scope*.

`typing.TypeVarTuples` and `typing.ParamSpecs` cannot have bounds or constraints.

All three flavors of type parameters can also have a *default value*, which is used when the type parameter is not explicitly provided. This is added by appending a single equals sign (`=`) followed by an expression. Like the bounds and constraints of type variables, the default value is not evaluated when the object is created, but only when the type parameter's `__default__` attribute is accessed. To this end, the default value is evaluated in a separate *annotation scope*. If no default value is specified for a type parameter, the `__default__` attribute is set to the special sentinel object `typing.NoDefault`.

The following example indicates the full set of allowed type parameter declarations:

```
def overly_generic[
    SimpleTypeVar,
    TypeVarWithDefault = int,
    TypeVarWithBound: int,
    TypeVarWithConstraints: (str, bytes),
    *SimpleTypeVarTuple = (int, float),
    **SimpleParamSpec = (str, bytearray),
] (
    a: SimpleTypeVar,
    b: TypeVarWithDefault,
    c: TypeVarWithBound,
    d: Callable[SimpleParamSpec, TypeVarWithConstraints],
    *e: SimpleTypeVarTuple,
): ...
```

### 8.10.1 Generic functions

Generic functions are declared as follows:

```
def func[T] (arg: T): ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_func():
    T = typing.TypeVar("T")
    def func(arg: T): ...
    func.__type_params__ = (T,)
    return func
func = TYPE_PARAMS_OF_func()
```

Here `annotation-def` indicates an *annotation scope*, which is not actually bound to any name at runtime. (One other liberty is taken in the translation: the syntax does not go through attribute access on the `typing` module, but creates an instance of `typing.TypeVar` directly.)

The annotations of generic functions are evaluated within the annotation scope used for declaring the type parameters, but the function's defaults and decorators are not.

The following example illustrates the scoping rules for these cases, as well as for additional flavors of type parameters:

```
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = some_default):
    ...
```

Except for the *lazy evaluation* of the `TypeVar` bound, this is equivalent to:

```
DEFAULT_OF_arg = some_default

annotation-def TYPE_PARAMS_OF_func():

    annotation-def BOUND_OF_T():
        return int
    # In reality, BOUND_OF_T() is evaluated only on demand.
    T = typing.TypeVar("T", bound=BOUND_OF_T())

    Ts = typing.TypeVarTuple("Ts")
    P = typing.ParamSpec("P")

    def func(*args: *Ts, arg: Callable[P, T] = DEFAULT_OF_arg):
        ...

    func.__type_params__ = (T, Ts, P)
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

The capitalized names like `DEFAULT_OF_arg` are not actually bound at runtime.

## 8.10.2 Generic classes

Generic classes are declared as follows:

```
class Bag[T]: ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(typing.Generic[T]):
        __type_params__ = (T,)
        ...
    return Bag
Bag = TYPE_PARAMS_OF_Bag()
```

Here again `annotation-def` (not a real keyword) indicates an *annotation scope*, and the name `TYPE_PARAMS_OF_Bag` is not actually bound at runtime.

Generic classes implicitly inherit from `typing.Generic`. The base classes and keyword arguments of generic classes are evaluated within the type scope for the type parameters, and decorators are evaluated outside that scope. This is illustrated by this example:

```
@decorator
class Bag(Base[T], arg=T): ...
```

This is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(Base[T], typing.Generic[T], arg=T):
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

### 8.10.3 Generic type aliases

The *type* statement can also be used to create a generic type alias:

```
type ListOrSet[T] = list[T] | set[T]
```

Except for the *lazy evaluation* of the value, this is equivalent to:

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")

    annotation-def VALUE_OF_ListOrSet():
        return list[T] | set[T]
    # In reality, the value is lazily evaluated
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_
    ↪params=(T,))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

Here, *annotation-def* (not a real keyword) indicates an *annotation scope*. The capitalized names like `TYPE_PARAMS_OF_ListOrSet` are not actually bound at runtime.

## 8.11 Annotations

Αλλάξε στην έκδοση 3.14: Annotations are now lazily evaluated by default.

Variables and function parameters may carry *annotations*, created by adding a colon after the name, followed by an expression:

```
x: annotation = 1
def f(param: annotation): ...
```

Functions may also carry a return annotation following an arrow:

```
def f() -> annotation: ...
```

Annotations are conventionally used for *type hints*, but this is not enforced by the language, and in general annotations may contain arbitrary expressions. The presence of annotations does not change the runtime semantics of the code, except if some mechanism is used that introspects and uses the annotations (such as `dataclasses` or `functools.singledispatch()`).

By default, annotations are lazily evaluated in an *annotation scope*. This means that they are not evaluated when the code containing the annotation is evaluated. Instead, the interpreter saves information that can be used to evaluate the annotation later if requested. The `annotationlib` module provides tools for evaluating annotations.

If the *future statement* from `__future__ import annotations` is present, all annotations are instead stored as strings:

```
>>> from __future__ import annotations
>>> def f(param: annotation): ...
>>> f.__annotations__
{'param': 'annotation'}
```

This future statement will be deprecated and removed in a future version of Python, but not before Python 3.13 reaches its end of life (see [PEP 749](#)). When it is used, introspection tools like `annotationlib.get_annotations()` and `typing.get_type_hints()` are less likely to be able to resolve annotations at runtime.

---

## 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) * ENDMARKER
```

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 | ENDMARKER
```

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* ENDMARKER
```

## 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 used here is the same as in the preceding docs, and is described in the *notation* section, except for an extra complication:

- ~ («cut»): ολοκλήρωση της τρέχουσας εναλλακτικής λύσης και αποτυχία του κανόνα ακόμα κι αν αυτή αποτύχει να αναλυθεί

```
# PEG grammar for Python

# ===== START OF THE GRAMMAR =====

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax.
→errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the
→invalid
#     rules will be executed.
#     - If the parser fails in the second phase with a generic syntax
→error, the
#     location of the generic failure of the first pass will be used.
→(this avoids
#     reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#     (like any rule in PEG).
#
# Grammar Syntax (see PEP 617 for more information):
#
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

# rule_name: expression
#   Optionally, a type can be included right after the rule name, which
#   specifies the return type of the C or Python function corresponding to
→the
#   rule:
# rule_name[return_type]: expression
#   If the return type is omitted, then a void * is returned in C and an
→Any in
#   Python.
# e1 e2
#   Match e1, then match e2.
# e1 | e2
#   Match e1 or e2.
#   The first alternative can also appear on the line after the rule name
→for
#   formatting purposes. In that case, a | must be used before the first
#   alternative, like so:
#       rule_name[return_type]:
#           | first_alt
#           | second_alt
# ( e )
#   Match e (allows also to use other operators in the group like '(e)*')
# [ e ] or e?
#   Optionally match e.
# e*
#   Match zero or more occurrences of e.
# e+
#   Match one or more occurrences of e.
# s.e+
#   Match one or more occurrences of e, separated by s. The generated
→parse tree
#   does not include the separator. This is otherwise identical to (e (s
→e)*).
# &e
#   Succeed if e can be parsed, without consuming any input.
# !e
#   Fail if e can be parsed, without consuming any input.
# ~
#   Commit to the current alternative, even if it fails to parse.
# &&e
#   Eager parse e. The parser will not backtrack and will immediately
#   fail with SyntaxError if e cannot be parsed.
#

# STARTING RULES
# =====

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

# GENERAL STATEMENTS
# =====

statements: statement+

```

(συνέχεια στην επόμενη σελίδα)



(συνεχίζεται από την προηγούμενη σελίδα)

```

statement:
    | compound_stmt
    | simple_stmts

single_compound_stmt:
    | compound_stmt

statement_newline:
    | single_compound_stmt NEWLINE
    | simple_stmts
    | NEWLINE
    | ENDMARKER

simple_stmts:
    | simple_stmt ';' NEWLINE # Not needed, there for speedup
    | ';' simple_stmt+ [';'] NEWLINE

# NOTE: assignment MUST precede expression, else parsing a simple_
→assignment
# will throw a SyntaxError.
simple_stmt:
    | assignment
    | type_alias
    | star_expressions
    | return_stmt
    | import_stmt
    | raise_stmt
    | pass_stmt
    | del_stmt
    | yield_stmt
    | assert_stmt
    | break_stmt
    | continue_stmt
    | global_stmt
    | nonlocal_stmt

compound_stmt:
    | function_def
    | if_stmt
    | class_def
    | with_stmt
    | for_stmt
    | try_stmt
    | while_stmt
    | match_stmt

# SIMPLE STATEMENTS
# =====

# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with
→'yield'
assignment:
    | NAME ':' expression ['=' annotated_rhs ]
    | '(' ( ' single_target ' )
        | single_subscript_attribute_target ) ':' expression ['='

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

→annotated_rhs ]
    | (star_targets '=' )+ annotated_rhs !=' [TYPE_COMMENT]
    | single_target augassign ~ annotated_rhs

annotated_rhs: yield_expr | star_expressions

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

return_stmt:
    | 'return' [star_expressions]

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

pass_stmt:
    | 'pass'

break_stmt:
    | 'break'

continue_stmt:
    | 'continue'

global_stmt: 'global' ', '.NAME+

nonlocal_stmt: 'nonlocal' ', '.NAME+

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

yield_stmt: yield_expr

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

import_stmt:
    | import_name
    | import_from

# Import statements
# -----

import_name: 'import' dotted_as_names

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

# note below: the ('.' | '...') is necessary because '...' is tokenized as_
→ELLIPSIS
import_from:
    | 'from' ('.' | '...')* dotted_name 'import' import_from_targets
    | 'from' ('.' | '...')+ 'import' import_from_targets
import_from_targets:
    | '(' import_from_as_names [',' ] ')'
    | import_from_as_names '!', '
    | '*'
import_from_as_names:
    | ','.import_from_as_name+
import_from_as_name:
    | NAME ['as' NAME ]

dotted_as_names:
    | ','.dotted_as_name+
dotted_as_name:
    | dotted_name ['as' NAME ]

dotted_name:
    | dotted_name '.' NAME
    | NAME

# COMPOUND STATEMENTS
# =====

# Common elements
# -----

block:
    | NEWLINE INDENT statements DEDENT
    | simple_stmts

decorators: ('@' named_expression NEWLINE )+

# Class definitions
# -----

class_def:
    | decorators class_def_raw
    | class_def_raw

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

# Function definitions
# -----

function_def:
    | decorators function_def_raw
    | function_def_raw

function_def_raw:
    | 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':'_
→[func_type_comment] block
    | 'async' 'def' NAME [type_params] '(' [params] ')' ['->' expression ]

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

→ ':' [func_type_comment] block

# Function parameters
# -----

params:
    | parameters

parameters:
    | slash_no_default param_no_default* param_with_default* [star_etc]
    | slash_with_default param_with_default* [star_etc]
    | param_no_default+ param_with_default* [star_etc]
    | param_with_default+ [star_etc]
    | star_etc

# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).

slash_no_default:
    | param_no_default+ '/' ','
    | param_no_default+ '/' &')'
slash_with_default:
    | param_no_default* param_with_default+ '/' ','
    | param_no_default* param_with_default+ '/' &')'

star_etc:
    | '*' param_no_default param_maybe_default* [kwds]
    | '*' param_no_default_star_annotation param_maybe_default* [kwds]
    | '*' ',' param_maybe_default+ [kwds]
    | kwds

kwds:
    | '*' param_no_default

# One parameter. This *includes* a following comma and type comment.
#
# There are three styles:
# - No default
# - With default
# - Maybe with default
#
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
#

param_no_default:
    | param ',' TYPE_COMMENT?
    | param TYPE_COMMENT? &')'
param_no_default_star_annotation:
    | param_star_annotation ',' TYPE_COMMENT?
    | param_star_annotation TYPE_COMMENT? &')'
param_with_default:
    | param default ',' TYPE_COMMENT?
    | param default TYPE_COMMENT? &')'

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

param_maybe_default:
    | param default? ',' TYPE_COMMENT?
    | param default? TYPE_COMMENT? &')'
param: NAME annotation?
param_star_annotation: NAME star_annotation
annotation: ':' expression
star_annotation: ':' star_expression
default: '=' expression | invalid_default

# If statement
# -----

if_stmt:
    | 'if' named_expression ':' block elif_stmt
    | 'if' named_expression ':' block [else_block]
elif_stmt:
    | 'elif' named_expression ':' block elif_stmt
    | 'elif' named_expression ':' block [else_block]
else_block:
    | 'else' ':' block

# While statement
# -----

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

# For statement
# -----

for_stmt:
    | 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block_
    ↳ [else_block]
    | 'async' 'for' star_targets 'in' ~ star_expressions ':' [TYPE_
    ↳ COMMENT] block [else_block]

# With statement
# -----

with_stmt:
    | 'with' '(' ','.with_item+ ',' '?' ')' ':' [TYPE_COMMENT] block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | 'async' 'with' '(' ','.with_item+ ',' '?' ')' ':' block
    | 'async' 'with' ','.with_item+ ':' [TYPE_COMMENT] block

with_item:
    | expression 'as' star_target &(',' | ')') | ':'
    | expression

# Try statement
# -----

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

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

# Except statement
# -----

except_block:
    | 'except' expression ':' block
    | 'except' expression 'as' NAME ':' block
    | 'except' expressions ':' block
    | 'except' ':' block
except_star_block:
    | 'except' '*' expression ':' block
    | 'except' '*' expression 'as' NAME ':' block
    | 'except' '*' expressions ':' block
finally_block:
    | 'finally' ':' block

# Match statement
# -----

match_stmt:
    | "match" subject_expr ':' NEWLINE INDENT case_block+ DEDENT

subject_expr:
    | star_named_expression ',' star_named_expressions?
    | named_expression

case_block:
    | "case" patterns guard? ':' block

guard: 'if' named_expression

patterns:
    | open_sequence_pattern
    | pattern

pattern:
    | as_pattern
    | or_pattern

as_pattern:
    | or_pattern 'as' pattern_capture_target

or_pattern:
    | '|' closed_pattern+

closed_pattern:
    | literal_pattern
    | capture_pattern
    | wildcard_pattern
    | value_pattern
    | group_pattern
    | sequence_pattern
    | mapping_pattern
    | class_pattern

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

# Literal patterns are used for equality and identity constraints
literal_pattern:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

# Literal expressions are used to restrict permitted mapping pattern keys
literal_expr:
    | signed_number !('+' | '-')
    | complex_number
    | strings
    | 'None'
    | 'True'
    | 'False'

complex_number:
    | signed_real_number '+' imaginary_number
    | signed_real_number '-' imaginary_number

signed_number:
    | NUMBER
    | '-' NUMBER

signed_real_number:
    | real_number
    | '-' real_number

real_number:
    | NUMBER

imaginary_number:
    | NUMBER

capture_pattern:
    | pattern_capture_target

pattern_capture_target:
    | !"_" NAME !('.' | '(' | '=')

wildcard_pattern:
    | "_"

value_pattern:
    | attr !('.' | '(' | '=')

attr:
    | name_or_attr '.' NAME

name_or_attr:
    | attr
    | NAME

group_pattern:

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

    | '(' pattern ')'

sequence_pattern:
    | '[' maybe_sequence_pattern? ']'
    | '(' open_sequence_pattern? ')'

open_sequence_pattern:
    | maybe_star_pattern ',' maybe_sequence_pattern?

maybe_sequence_pattern:
    | ','.maybe_star_pattern+ ','?

maybe_star_pattern:
    | star_pattern
    | pattern

star_pattern:
    | '*' pattern_capture_target
    | '*' wildcard_pattern

mapping_pattern:
    | '{' '}'
    | '{' double_star_pattern ','? '}'
    | '{' items_pattern ',' double_star_pattern ','? '}'
    | '{' items_pattern ','? '}'

items_pattern:
    | ','.key_value_pattern+

key_value_pattern:
    | (literal_expr | attr) ':' pattern

double_star_pattern:
    | '**' pattern_capture_target

class_pattern:
    | name_or_attr '(' ' )'
    | name_or_attr '(' positional_patterns ','? ' )'
    | name_or_attr '(' keyword_patterns ','? ' )'
    | name_or_attr '(' positional_patterns ',' keyword_patterns ','? ' )'

positional_patterns:
    | ','.pattern+

keyword_patterns:
    | ','.keyword_pattern+

keyword_pattern:
    | NAME '=' pattern

# Type statement
# -----

type_alias:
    | "type" NAME [type_params] '=' expression

```

(συνέχεια στην επόμενη σελίδα)



(συνεχίζεται από την προηγούμενη σελίδα)

```

# Type parameter declaration
# -----

type_params:
    | '[' type_param_seq ']'

type_param_seq: ','.type_param+ [',']

type_param:
    | NAME [type_param_bound] [type_param_default]
    | '*' NAME [type_param_starred_default]
    | '**' NAME [type_param_default]

type_param_bound: ':' expression
type_param_default: '=' expression
type_param_starred_default: '=' star_expression

# EXPRESSIONS
# -----

expressions:
    | expression (',' expression )+ [',']
    | expression ','
    | expression

expression:
    | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef

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

star_expressions:
    | star_expression (',' star_expression )+ [',']
    | star_expression ','
    | star_expression

star_expression:
    | '*' bitwise_or
    | expression

star_named_expressions: ','.star_named_expression+ [',']

star_named_expression:
    | '*' bitwise_or
    | named_expression

assignment_expression:
    | NAME ':=' ~ expression

named_expression:
    | assignment_expression
    | expression !':='

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

disjunction:
    | conjunction ('or' conjunction )+
    | conjunction

conjunction:
    | inversion ('and' inversion )+
    | inversion

inversion:
    | 'not' inversion
    | comparison

# Comparison operators
# -----

comparison:
    | bitwise_or compare_op_bitwise_or_pair+
    | bitwise_or

compare_op_bitwise_or_pair:
    | eq_bitwise_or
    | noteq_bitwise_or
    | lte_bitwise_or
    | lt_bitwise_or
    | gte_bitwise_or
    | gt_bitwise_or
    | notin_bitwise_or
    | in_bitwise_or
    | isnot_bitwise_or
    | is_bitwise_or

eq_bitwise_or: '=' bitwise_or
noteq_bitwise_or:
    | ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or
lt_bitwise_or: '<' bitwise_or
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or

# Bitwise operators
# -----

bitwise_or:
    | bitwise_or '|' bitwise_xor
    | bitwise_xor

bitwise_xor:
    | bitwise_xor '^' bitwise_and
    | bitwise_and

bitwise_and:
    | bitwise_and '&' shift_expr

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

    | shift_expr

shift_expr:
    | shift_expr '<<' sum
    | shift_expr '>>' sum
    | sum

# Arithmetic operators
# -----

sum:
    | sum '+' term
    | sum '-' term
    | term

term:
    | term '*' factor
    | term '/' factor
    | term '//' factor
    | term '%' factor
    | term '@' factor
    | factor

factor:
    | '+' factor
    | '-' factor
    | '~' factor
    | power

power:
    | await_primary '**' factor
    | await_primary

# Primary elements
# -----

# Primary elements are things like "obj.something.something",
# → "obj[something]", "obj(something)", "obj" ...

await_primary:
    | 'await' primary
    | primary

primary:
    | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom

slices:
    | slice '!', ','
    | '!', '.' (slice | starred_expression)+ ['!', ',']

slice:
    | [expression] ':' [expression] [':' [expression] ]

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

    | named_expression

atom:
    | NAME
    | 'True'
    | 'False'
    | 'None'
    | strings
    | NUMBER
    | (tuple | group | genexp)
    | (list | listcomp)
    | (dict | set | dictcomp | setcomp)
    | '...'

group:
    | '(' (yield_expr | named_expression) ')'

# Lambda functions
# -----

lambdef:
    | 'lambda' [lambda_params] ':' expression

lambda_params:
    | lambda_parameters

# lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis. (For more, see parameters above.)
#
lambda_parameters:
    | lambda_slash_no_default lambda_param_no_default* lambda_param_with_
    ↳ default* [lambda_star_etc]
    | lambda_slash_with_default lambda_param_with_default* [lambda_star_
    ↳ etc]
    | lambda_param_no_default+ lambda_param_with_default* [lambda_star_etc]
    | lambda_param_with_default+ [lambda_star_etc]
    | lambda_star_etc

lambda_slash_no_default:
    | lambda_param_no_default+ '/' ','
    | lambda_param_no_default+ '/' & ':'

lambda_slash_with_default:
    | lambda_param_no_default* lambda_param_with_default+ '/' ','
    | lambda_param_no_default* lambda_param_with_default+ '/' & ':'

lambda_star_etc:
    | '*' lambda_param_no_default lambda_param_maybe_default* [lambda_kwds]
    | '*' ',' lambda_param_maybe_default+ [lambda_kwds]
    | lambda_kwds

lambda_kwds:
    | '**' lambda_param_no_default

lambda_param_no_default:

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

    | lambda_param ','
    | lambda_param &':'
lambda_param_with_default:
    | lambda_param default ','
    | lambda_param default &':'
lambda_param_maybe_default:
    | lambda_param default? ','
    | lambda_param default? &':'
lambda_param: NAME

# LITERALS
# =====

fstring_middle:
    | fstring_replacement_field
    | FString_MIDDLE
fstring_replacement_field:
    | '{' annotated_rhs '='? [fstring_conversion] [fstring_full_format_
↪spec] '}'
fstring_conversion:
    | "!" NAME
fstring_full_format_spec:
    | ':' fstring_format_spec*
fstring_format_spec:
    | FString_MIDDLE
    | fstring_replacement_field
fstring:
    | FString_START fstring_middle* FString_END

tstring_format_spec_replacement_field:
    | '{' annotated_rhs '='? [fstring_conversion] [tstring_full_format_
↪spec] '}'
tstring_format_spec:
    | TString_MIDDLE
    | tstring_format_spec_replacement_field
tstring_full_format_spec:
    | ':' tstring_format_spec*
tstring_replacement_field:
    | '{' annotated_rhs '='? [fstring_conversion] [tstring_full_format_
↪spec] '}'
tstring_middle:
    | tstring_replacement_field
    | TString_MIDDLE
tstring:
    | TString_START tstring_middle* TString_END

string: STRING
strings:
    | (fstring|string)+
    | tstring+

list:
    | '[' [star_named_expressions] ']'

tuple:
    | '(' [star_named_expression ',' [star_named_expressions] ] ')'

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

set: '{' star_named_expressions '}'

# Dicts
# -----

dict:
    | '{' [double_starred_kvpairs] '}'

double_starred_kvpairs: ','.double_starred_kvpair+ [',' ]

double_starred_kvpair:
    | '**' bitwise_or
    | kvpair

kvpair: expression ':' expression

# Comprehensions & Generators
# -----

for_if_clauses:
    | for_if_clause+

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

listcomp:
    | '[' named_expression for_if_clauses ']'

setcomp:
    | '{' named_expression for_if_clauses '}'

genexp:
    | '(' ( assignment_expression | expression !':=' ) for_if_clauses ')'

dictcomp:
    | '{' kvpair for_if_clauses '}'

# FUNCTION CALL ARGUMENTS
# =====

arguments:
    | args [',' &')'

args:
    | ','. (starred_expression | ( assignment_expression | expression !':='
→ ') !':=' )+ [',' kwargs ]
    | kwargs

kwargs:
    | ','. karg_or_starred+ ','. karg_or_double_starred+
    | ','. karg_or_starred+
    | ','. karg_or_double_starred+

starred_expression:

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

    | '*' expression

kwarg_or_starred:
    | NAME '=' expression
    | starred_expression

kwarg_or_double_starred:
    | NAME '=' expression
    | '**' expression

# ASSIGNMENT TARGETS
# =====

# Generic targets
# -----

# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
    | star_target !','
    | star_target '(' star_target ')' * [',' ]

star_targets_list_seq: ',' star_target+ [',' ]

star_targets_tuple_seq:
    | star_target '(' star_target ')' + [',' ]
    | star_target ','

star_target:
    | '*' (!'*' star_target)
    | target_with_star_atom

target_with_star_atom:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom

star_atom:
    | NAME
    | '(' target_with_star_atom ')'
    | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'

single_target:
    | single_subscript_attribute_target
    | NAME
    | '(' single_target ')'

single_subscript_attribute_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead

t_primary:
    | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
    | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

    | atom &t_lookahead

t_lookahead: '(' | '[' | '.'

# Targets for del statements
# -----

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

del_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | del_t_atom

del_t_atom:
    | NAME
    | '(' del_target ')'
    | '(' [del_targets] ')'
    | '[' [del_targets] ']'

# TYPING ELEMENTS
# -----

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

func_type_comment:
    | NEWLINE TYPE_COMMENT &(NEWLINE INDENT) # Must be followed by_
    ↪indented block
    | TYPE_COMMENT

# ===== END OF THE GRAMMAR =====

# ===== START OF INVALID RULES =====

```



>>>

Η προεπιλεγμένη Python εντολή του *interactive* shell. Συχνά εμφανίζεται για παραδείγματα κώδικα που μπορούν να εκτελεστούν διαδραστικά στον interpreter.

...

Μπορεί να αναφέρεται σε:

- Η προεπιλεγμένη Python εντολή του *interactive* 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.

#### 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`.

#### συνάρτηση annotate

Μια συνάρτηση που μπορεί να κληθεί για να ανακτήσει το *annotations* ενός αντικειμένου. Αυτή η συνάρτηση είναι προσβάσιμη ως το χαρακτηριστικό `__annotate__` των συναρτήσεων, των κλάσεων και των modules. Οι συναρτήσεις `annotate` είναι ένα υποσύνολο του *evaluate functions*.

#### 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 μπορούν να ανακτηθούν καλώντας την εντολή `annotationlib.get_annotations()` σε modules, κλάσεις και συναρτήσεις, αντίστοιχα.

Βλ. τα *variable annotation*, *function annotation*, **PEP 484**, **PEP 526** και **PEP 649**, τα οποία περιγράφουν την λειτουργικότητα. Δείτε επίσης τα annotations-howto για τις βέλτιστες πρακτικές δουλεύοντας με

annotations.

### 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 αρχή 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 γλωσσarium για το *parameter*, την FAQ ερώτηση στο η διαφορά μεταξύ ορισμάτων και παραμέτρων, και [PEP 362](#).

### ασύγχρονος διαχειριστής context

Ένα αντικείμενο που ελέγχει το ορατό περιβάλλον σε μια δήλωση *async with* ορίζοντας τις μεθόδους `__aenter__()` και `__aexit__()`. Που εισήχθη από [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

An object created by an *asynchronous generator* function.

Αυτός είναι ένας *asynchronous iterator* που όταν καλείται χρησιμοποιώντας την μέθοδο `__anext__()` επιστρέφει ένα αναμενόμενο αντικείμενο που θα εκτελέσει στο σώμα της συνάρτησης του ασύγχρονου generator μέχρι την επόμενη *yield* έκφραση.

Κάθε *yield* αναστέλλει προσωρινά την επεξεργασία, θυμάται την κατάσταση εκτέλεσης (συμπεριλαμβανομένων των τοπικών μεταβλητών και των δηλώσεων *try* σε εκκρεμότητα). Όταν ο *ασύγχρονος generator iterator* συνεχίσει αποτελεσματικά με άλλο αναμενόμενο που επιστρέφεται από `:pep: 492()` και [PEP 525](#).

### ασύγχρονος iterable

Ένα αντικείμενο, που μπορεί να χρησιμοποιηθεί σε μια δήλωση *async for*. Πρέπει να επιστρέφει ένα *asynchronous iterator* από την μέθοδο `__aiter__()`. Που εισήχθη από [PEP 492](#).

### ασύγχρονος iterator

Ένα αντικείμενο που υλοποιεί τις μεθόδους `__aiter__()` και `__anext__()`. Η μέθοδος `__anext__()` πρέπει να επιστρέφει ένα *awaitable* αντικείμενο. Το *async for* επιλύει τα αναμενόμενα που επιστρέφονται από τη μέθοδο `__anext__()` ενός ασύγχρονου iterator έως ότου εγείρει μια εξαίρεση `StopAsyncIteration`. Εισήχθη από [PEP 492](#).

**κατάσταση συνδεδεμένου νήματος**

Ένα *thread state* που είναι ενεργή για το τρέχον νήμα του λειτουργικού συστήματος.

Όταν επισυνάπτεται ένας *thread state*, το νήμα του λειτουργικού συστήματος έχει πρόσβαση στο πλήρες Python C API και μπορεί να καλέσει με ασφάλεια τον διερμηνέα bytecode.

Εκτός εάν μια συνάρτηση αναφέρει ρητά το αντίθετο, η προσπάθεια κλήσης του C API χωρίς μια συνημμένη κατάσταση νήματος θα οδηγήσει ένα μοιραίο σφάλμα ή σε απροσδιόριστη συμπεριφορά. Μια κατάσταση νήματος μπορεί να συνδεθεί και να αποσυνδεθεί ρητά από τον χρήστη μέσω του C API ή έμμεσα από τον χρόνο εκτέλεσης, συμπεριλαμβανομένων των κλήσεων αποκλεισμού C και από τον διερμηνέα bytecode μεταξύ των κλήσεων.

Στις περισσότερες εκδόσεις της Python, η ύπαρξη μιας κατάσταση συνδεδεμένου νήματος υπονοεί ότι ο καλών διατηρεί την *GIL* για τον τρέχοντα διερμηνέα, επομένως μόνο ένα νήμα λειτουργικού συστήματος μπορεί να έχει μια κατάσταση συνδεδεμένου νήματος σε μια δεδομένη στιγμή. Στις εκδόσεις *free-threaded* της Python, τα νήματα μπορούν να διατηρούν ταυτόχρονα μια κατάσταση συνδεδεμένου νήματος, επιτρέποντας την πραγματική παραλληλία του διερμηνέα bytecode.

**χαρακτηριστικό**

Μια τιμή που σχετίζεται με ένα αντικείμενο που συνήθως αναφέρεται με όνομα χρησιμοποιώντας εκφράσεις με κουκκίδες. Για παράδειγμα, εάν ένα αντικείμενο *o* έχει ένα χαρακτηριστικό *a* θα αναφέρεται ως *o.a*.

Είναι δυνατό να δώσουμε σε ένα αντικείμενο ένα χαρακτηριστικό που το όνομα του δεν είναι αναγνωριστικό όπως ορίζεται από *Names (identifiers and keywords)*, για παράδειγμα χρησιμοποιώντας `setattr()`, αν επιτρέπεται από το αντικείμενο. Ένα τέτοιο χαρακτηριστικό δεν θα είναι προσβάσιμο χρησιμοποιώντας τις τελείες, και αντί αυτού θα πρέπει να ανακτηθεί χρησιμοποιώντας `getattr()`.

**awaitable**

Ένα αντικείμενο που μπορεί να χρησιμοποιηθεί στην έκφραση *await*. Μπορεί να είναι *coroutine* ή ένα αντικείμενο με μια `__await__()` μέθοδο. Βλ. επίσης [PEP 492](#).

**BDFL**

Ακρωνύμιο του *Benevolent Dictator For Life*, καλοκάγαθος δικτάτορας της ζωής, δηλαδή [Guido van Rossum](#), ο δημιουργός της Python.

**δυναδικό αρχείο**

Ένα *file object* ικανό να διαβάσει και να γράφει *δυναδικού τύπου αντικείμενα*. Παραδείγματα δυναδικών αρχείων είναι αρχεία που ανοίγουν σε δυναδική λειτουργία ('rb', 'wb' ή 'rb+'), `sys.stdin.buffer`, `sys.stdout.buffer`, και στιγμιотύπων των `io.BytesIO` και `gzip.GzipFile`.

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

**δανεική αναφορά**

Στο C API της Python, μια δανεική αναφορά είναι μια αναφορά σε ένα αντικείμενο, όπου ο κώδικας που χρησιμοποιεί το αντικείμενο δεν κατέχει την αναφορά. Γίνεται ένας αχρησιμοποίητος δείκτης εάν το αντικείμενο καταστραφεί. Για παράδειγμα, μια διαδικασία *garbage collection* μπορεί να αφαιρέσει το τελευταίο *strong reference* από το αντικείμενο και έτσι να το καταστρέψει.

Συνίσταται η κλήση του `Py_INCREF()` στο *δανεική αναφορά* με σκοπό να μετατραπεί σε ένα *ισχυρή αναφορά* επιτόπου, εκτός όταν το αντικείμενο δεν μπορεί να καταστραφεί πριν από την τελευταία χρήση της δανεικής αναφοράς. Η συνάρτηση `Py_NewRef()` μπορεί να χρησιμοποιηθεί ώστε να δημιουργηθεί ένα *ισχυρή αναφορά*.

**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 δυναδικού τύπου (bytes-like) αντικείμενα μπορούν να χρησιμοποιηθούν για διάφορες λειτουργίες που διαχειρίζονται δυναδικά δεδομένα" αυτά περιλαμβάνουν συμπίεση αποθήκευση σε δυναδικό αρχείο και αποστολή μέσω socket.

Ορισμένες λειτουργίες χρειάζονται τα δυναδικά δεδομένα να είναι μεταβλητά. Η τεκμηρίωση συχνά αναφέρεται σε αυτά ως «δυναδικά αντικείμενα ανάγνωσης-εγγραφής» (read-write bytes-like objects). Παραδείγματα μεταβλητών αντικειμένων προσωρινής αποθήκευσης περιέχουν `bytearray` και ένα

`memoryview` ενός `bytearray`. Άλλες λειτουργίες απαιτούν την αποθήκευση των δυαδικών δεδομένων 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 κώδικας 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 *bytecode*). 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`.

### callable

Ένα callable είναι ένα αντικείμενο που μπορεί να καλεστεί, πιθανά με ένα σύνολο ορισμάτων (βλ. *argument*), με την παρακάτω σύνταξη:

```
callable(argument1, argument2, argumentN)
```

Μια *function*, και κατ' επέκταση μια *method* είναι callable. Ένα στιγμιότυπο μια κλάσης που υλοποιεί τη μέθοδο `__call__()` είναι επίσης callable.

### 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).

### μεταβλητή κλεισίματος

Ένας *free variable* που αναφέρεται από ένα *nested scope* και ορίζεται σε μια εξωτερική περιοχή, αντί να επιλύεται δυναμικά κατά την εκτέλεση από τα καθολικά ή ενσωματωμένα namespaces. Μπορεί να δηλωθεί ρητά με τη δεσμευμένη λέξη-κλειδί *nonlocal* ώστε να επιτραπεί η εγγραφή, ή να θεωρηθεί ότι ορίζεται έμμεσα όταν η μεταβλητή χρησιμοποιείται μόνο για ανάγνωση.

Για παράδειγμα, η συνάρτηση `inner` του παρακάτω κώδικα, τόσο η `x` όσο και η `print` είναι *free variables*, αλλά μόνο η `x` είναι μια μεταβλητή κλεισίματος:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Λόγο του χαρακτηριστικού `codeobject.co_freevars` (το οποίο, παρά την ονομασία του, περιλαμβάνει μόνο τα ονόματα των μεταβλητών κλεισίματος και όχι όλες τις αναφερόμενες ελεύθερες μεταβλητές), χρησιμοποιείται μερικές φορές ο πιο γενικός όρος *free variable* ακόμη και όταν γίνεται ειδική αναφορά σε μεταβλητές κλεισίματος.

### μυγαδικός 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 μυγαδικούς

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.

### context

Αυτό ο όρος έχει διαφορετικές σημασίες ανάλογα με το πού και πώς χρησιμοποιείται. Μερικές κοινές έννοιες:

- Η προσωρινή κατάσταση ή το περιβάλλον που δημιουργείται από έναν *context manager* μέσω μιας δήλωσης *with*.
- Το σύνολο των δεσμευμένων κλειδιού-τιμής που σχετίζονται με ένα συγκεκριμένο αντικείμενο `contextvars.Context` και προσπελάζονται μέσω αντικειμένων `ContextVar`. Βλ. επίσης *context variable*.
- Ένα αντικείμενο `contextvars.Context`. Βλ. επίσης *current context*.

### πρωτόκολλο διαχείρισης περιβάλλοντος

Οι μέθοδοι `__enter__()` και `__exit__()` καλούνται από τη δήλωση *with*. Βλ. **PEP 343**.

### διαχειριστής context

Ένα αντικείμενο που υλοποιεί το *context management protocol* και ελέγχει το περιβάλλον που είσαι ορατό μέσα σε μια δήλωση *with*. Βλ. **PEP 343**.

### context μεταβλητή

Μια μεταβλητή της οποίας η τιμή εξαρτάται από το ποιο είναι το *current context*. Οι τιμές προσπελάζονται μέσω των αντικειμένων `contextvars.ContextVar`. Οι μεταβλητές συμφραζόμενων χρησιμοποιούνται κυρίως για να απομονώσουν την κατάσταση μεταξύ ταυτόχρονων ασύγχρονων εργασιών.

### contiguous

Ένα buffer θεωρείται contiguous ακριβώς εάν είναι είτε *C-contiguous* είτε *Fortran contiguous*. 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 υλοποιηθούν 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](http://python.org). Ο όρος «CPython» χρησιμοποιείται όταν είναι απαραίτητο για την διάκριση αυτής της υλοποίησης από άλλες όπως η *Jython* ή η *IronPython*.

### τρέχον πλαίσιο

Το *context* (`contextvars.Context` αντικείμενο) που χρησιμοποιείται αυτή τη στιγμή από τα αντικείμενα `ContextVar` για να προσπελάσει (να πάρει ή να ορίσει) τις τιμές των *context variables*. Κάθε νήμα έχει το δικό του τρέχον συμφραζόμενο Τα πλαίσια για την εκτέλεση ασύγχρονων εργασιών (βλ. *asyncio*) συνδέουν κάθε εργασία με ένα συμφραζόμενο, το οποίο γίνεται το τρέχον συμφραζόμενο όποτε η εργασία ξεκινά ή συνεχίζει την εκτέλεση.

### κυκλική απομόνωση

Μια υποομάδα ενός ή περισσότερων αντικειμένων που αναφέρονται μεταξύ τους σχηματίζοντας έναν κύκλο αναφορών, αλλά δεν αναφέρονται από άλλα αντικείμενα εκτός της ομάδας. Ο σκοπός του *cyclic*



*garbage collector* είναι να εντοπίζει αυτές τις ομάδες και να σπάει τον κύκλο αναφορών ώστε να μπορεί να αποδεσμευτεί η μνήμη.

### 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

Κάθε αντικείμενο που ορίζει τις μεθόδους `__get__()`, `__set__()`, ή `__delete__()`. Όταν ένα χαρακτηριστικό κλάσης είναι descriptor, η ειδική δεσμευτική του συμπεριφορά ενεργοποιείται κατά την αναζήτηση χαρακτηριστικών. Κανονικά, χρησιμοποιώντας `a.b` για να λάβετε, να ορίσετε ή να διαγράψετε ένα χαρακτηριστικό αναζητά το αντικείμενο με το όνομα `b` στο λεξικό της κλάσης για `a`, αλλά εάν το `b` είναι descriptor, καλείται η αντίστοιχη μέθοδος descriptor. Η κατανόηση των descriptors είναι το κλειδί για την καλύτερη κατανόηση της Python γιατί αυτό αποτελεί την βάση για πολλά χαρακτηριστικά όπως συναρτήσεις, μεθόδους, ιδιότητες, μέθοδοι κλάσης στατικές μέθοδοι, και αναφορά σε σούπερ κλάσεις.

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

### λεξικό

Ένα προσαρτησιακό πίνακα, όπου αυθαίρετα κλειδιά αντιστοιχίζονται σε τιμές. Τα κλειδιά μπορεί να είναι οποιοδήποτε αντικείμενο με μεθόδους `__hash__()` και `__eq__()`. Ονομάζεται ως hash στο 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

Μια literal συμβολοσειρά που εμφανίζεται ως η πρώτη έκφραση σε μια κλάση, συνάρτηση ή module. Ενώ αγνοείται κατά την εκτέλεση της σουίτας, αναγνωρίζεται από τον μεταγλωττιστή και τοποθετείται στο χαρακτηριστικό `__doc__` της κλάσης, της συνάρτησης ή του module που περικλείει. Δεδομένου ότι είναι διαθέσιμο μέσω ενδοσκόπησης, το κανονικό μέρος για την τεκμηρίωση του αντικειμένου.

### 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*.

### dunder

An informal short-hand for «double underscore», used when talking about a *special method*. For example, `__init__` is often pronounced «dunder init».

### 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 αποδοχθεί 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.

### αξιολόγηση συνάρτησης

Μια συνάρτηση που μπορεί να κληθεί για να αξιολογήσει ένα αδρανές χαρακτηριστικό ενός αντικειμένου, όπως η τιμή των ψευδωνύμων τύπου που δημιουργούνται με την πρόταση *type*.

### 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. Υπάρχουν επίσης *statements* 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

#### f-strings

String literals prefixed with `f` or `F` are commonly called «f-strings» which is short for *formatted string literals*. See also [PEP 498](#).

### αντικείμενο αρχείου

Ένα αντικείμενο που εκθέτει ένα API προσανατολισμένο σε αρχείο (με μεθόδους όπως `read()` ή `write()`) σε έναν υποκείμενο πόρο. Ανάλογα με τον τρόπο που δημιουργήθηκε, ένα αντικείμενο αρχείου μπορεί να μεσολαβήσει στην πρόσβαση σε ένα πραγματικό αρχείο στο δίσκο ή σε άλλο τύπο συσκευής αποθήκευσης ή επικοινωνίας (για παράδειγμα τυπική είσοδος/ έξοδος, in-memory buffers, sockets, pipes, κλπ.). Αντικείμενο αρχείου ονομάζονται επίσης *file-like objects* ή *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*.

### κωδικοποίηση συστήματος αρχείων και χειριστής σφαλμάτων

Η κωδικοποίηση και ο χειριστής σφαλμάτων χρησιμοποιείται από την Python για την αποκωδικοποίηση των bytes από το λειτουργικό σύστημα και την κωδικοποίηση σε Unicode για το λειτουργικό σύστημα.

Η κωδικοποίηση συστήματος αρχείων μπορεί να εγγραφεί την επιτυχημένη αποκωδικοποίηση όλων των bytes κάτω από 128. Εάν η κωδικοποίηση συστήματος αρχείων δεν παρέχει αυτήν την εγγύηση, οι συναρτήσεις API μπορούν να εγείρουν ένα `UnicodeError`.

Οι συναρτήσεις `sys.getfilesystemencoding()` και `sys.getfilesystemencodeerrors()` μπορούν να χρησιμοποιηθούν για να λάβετε την κωδικοποίηση του συστήματος αρχείων και του χειριστή σφαλμάτων.

Ο *filesystem encoding and error handler* διαμορφώνονται κατά την εκκίνηση της Python από τη συνάρτηση `PyConfig_Read()` βλ. `filesystem_encoding` και `filesystem_errors` μέλη του `PyConfig`.

Βλ. επίσης το *locale encoding*.

### finder

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

Υπάρχουν δύο τύποι finder: *finders μετα διαδρομής* για χρήση με `sys.meta_path`, και *finders εισόδου διαδρομής* για χρήση με `sys.path_hooks`.

Βλ. *Finders and loaders* και `importlib` για περισσότερες λεπτομέρειες.

### ακέραια διαίρεση

Η μαθηματική διαίρεση που στρογγυλοποιεί προς τα κάτω στον κοντινότερο ακέραιο. Ο τελεστής ακεράιας 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**.

### δωρεάν νήμα

Ένα μοντέλο νημάτων όπου πολλά νήματα μπορούν να εκτελούν Python bytecode ταυτόχρονα μέσα στον ίδιο διεργαστή. Αυτό έρχεται σε αντίθεση με το *global interpreter lock*, το οποίο επιτρέπει σε ένα μόνο νήμα να εκτελεί Python bytecode κάθε φορά. Δείτε το **PEP 703**.

### δωρεάν μεταβλητή

Τυπικά, όπως ορίζεται στο *language execution model*, μια ελεύθερη μεταβλητή είναι οποιαδήποτε μεταβλητή χρησιμοποιείται σε ένα namespace που δεν είναι τοπική μεταβλητή σε εκείνο το namespace. Δείτε το *closure variable* για παράδειγμα. Πρακτικά, λόγω του ονόματος του χαρακτηριστικού `codeobject.co_freevars`, ο όρος χρησιμοποιείται επίσης μερικές φορές ως συνώνυμο της *closure variable*.

### συνάρτηση

Μια σειρά από δηλώσεις που επιστρέφουν κάποια τιμή σε αυτόν που την κάλεσε. Σε αυτές μπορούν να περασθούν 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*.

Βλ. *variable annotation* και **PEP 484**, που περιγράφει αυτή την λειτουργικότητα. Επίσης βλ. *annotations-howto* για τις καλύτερες πρακτικές δουλεύοντας με annotations.

### \_\_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 τότε μια νέα δυνατότητα προστέθηκε για πρώτη φορά στην γλώσσα και τότε θα γίνει (ή έγινε) η προεπιλογή:

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

### συλλογή απορριμμάτων

Η διαδικασία απελευθέρωσης της μνήμης όταν δεν χρησιμοποιείται άλλο. Η Python εκτελεί συλλογή απορριμμάτων μέσω καταμέτρησης αναφορών και ενός κυκλικού συλλέκτη σκουπιδιών που είναι σε



θέση να ανιχνεύει και να σπάει τους κύκλους αναφοράς. Ο συλλέκτης απορριμμάτων 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*.

Κάθε *yield* αναστέλλει προσωρινά την επεξεργασία, θυμάται την κατάσταση εκτέλεσης (συμπεριλαμβανομένων των τοπικών μεταβλητών και των δηλώσεων δοκιμής σε εκκρεμότητα). Όταν ο *generator iterator* συνεχίσει, συνεχίζει από εκεί που σταμάτησε (σε αντίθεση με τις συναρτήσεις που ξεκινούν από την αρχή σε κάθε επίκληση).

### generator έκφραση

Μια *expression* 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.

Ωστόσο, ορισμένες λειτουργικές μονάδες επέκτασης, είτε τυπικές είτε τρίτων, έχουν σχεδιαστεί έτσι ώστε να απελευθερώνουν το GIL όταν εκτελούν εργασίες εντατικών υπολογισμών όπως συμπίεση ή κατακερματισμός. Επίσης, το GIL απελευθερώνεται πάντα όταν εκτελείτε I/O.

Από την έκδοση Python 3.13, ο GIL μπορεί να απενεργοποιηθεί χρησιμοποιώντας τη ρύθμιση `--disable-gil` κατά τη διαμόρφωση της κατασκευής. Μετά την κατασκευή της Python με αυτήν με αυτήν την επιλογή, ο κώδικας πρέπει να εκτελείται με την επιλογή `-X gil=0` ή αφού ρυθμιστεί η μεταβλητή περιβάλλοντος `PYTHON_GIL=0`. Αυτή η δυνατότητα επιτρέπει βελτιωμένη απόδοση για

εφαρμογές πολλαπλών νημάτων και διευκολύνει τη χρήση των επεξεργαστών πολλαπλών πυρήνων με αποδοτικό τρόπο. Για περισσότερες λεπτομέρειες, δείτε το [PEP 703](#).

Σε προηγούμενες εκδόσεις του C API της Python, μια συνάρτηση μπορεί να δηλώνει ότι απαιτεί την τήρηση του GIL για να χρησιμοποιηθεί. Αυτό αναφέρεται στην ύπαρξη μιας κατάστασης *attached thread state*.

### 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

Ένα αντικείμενο είναι *hashable* εάν έχει μια τιμή κατακερματισμού που δεν αλλάζει ποτέ κατά τη διάρκεια της ζωής του (χρειάζεται μια μέθοδο `__hash__()`), και μπορεί να συγκριθεί με άλλα αντικείμενα (χρειάζεται μια μέθοδο `__eq__()`). Τα *hashable* αντικείμενα που συγκρίνονται ως προς την ισότητα τους πρέπει να έχουν την ίδια τιμή κατακερματισμού.

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 frozensets) 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

Ένα ολοκληρωμένο περιβάλλον ανάπτυξης και μάθησης για την Python. `idle` είναι ένα βασικό περιβάλλον επεξεργασίας και διερμηνέα που συνοδεύεται από την τυπική διανομή της Python.

### Αθάνατο

*Αθάνατα αντικείμενα* είναι μια λεπτομέρεια υλοποίησης της CPython που εισήχθη στην [PEP 683](#).

Εάν ένα αντικείμενο είναι αθάνατο, ο *πλήθος αναφορές* του δεν τροποποιείται, και επομένως δεν εκχωρείται ποτέ ενώ εκτελείται ο διερμηνέας. Για παράδειγμα, `True` και `None` είναι αθάνατα στην CPython.

Τα αθάνατα αντικείμενα μπορούν να αναγνωριστούν μέσω της `sys._is_immortal()`, ή μέσω της `PyUnstable_IsImmortal()` στο C API.

### 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 αποτελέσματα.

Απλώς εκκινήστε την python χωρίς ορίσματα (πιθανώς επιλέγοντας το από το κύριο μενού του υπολογιστή σας). Αποτελεί έναν αποδοτικό τρόπο για να δοκιμάσετε νέες ιδέες ή να εξετάσετε modules και πακέτα (θυμηθείτε `help(x)`). Για περισσότερα σχετικά με τη διαδραστική λειτουργία, δείτε `tut-interac`.

### 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

Ένα αντικείμενο ικανό να επιστρέψει τα μέλη του ένα κάθε φορά. Παραδείγματα iterables περιλαμβάνουν όλους του τύπους ακολουθιών (όπως `list`, `str`, και `tuple`) και μερικούς τύπους μη ακολουθίας όπως `dict`, *αντικείμενο αρχείου*, και αντικείμενα οποιονδήποτε κλάσεων που μπορούν να οριστούν με μια μέθοδο `__iter__()` ή με μία μέθοδο `__getitem__()` που υλοποιεί τη σημασιολογία *sequence*.

Τα iterables μπορούν να χρησιμοποιηθούν σε ένα `for` βρόχο και σε πολλά άλλα σημεία όπου χρειάζεται μια ακολουθία (`zip()`, `map()`, ...). Όταν ένα iterable αντικείμενο μεταβιβάζεται ως όρισμα στην ενσωματωμένη συνάρτηση `iter()`, επιστρέφει έναν iterator για αντικείμενο. Αυτός ο iterator είναι καλός για ένα πέρασμα από ένα σύνολο τιμών. Όταν χρησιμοποιείται επαναληπτικά, συνήθως δεν είναι απαραίτητο να καλέσετε το `iter()` ή να ασχοληθείτε μόνοι σας με αντικείμενα iterator. Η δήλωση `for` το κάνει αυτόματα για εσάς, δημιουργώντας μια προσωρινή μεταβλητή χωρίς όνομα για να κρατά τον iterator για την διάρκεια του βρόχου. Βλ. επίσης *iterator*, *sequence*, και *generator*.

### iterator

Ένα αντικείμενο που αντιπροσωπεύει μια ροή δεδομένων. Επαναλαμβανόμενες κλήσεις προς τη μέθοδο `__next__()` του iterator (ή μεταβίβαση του στην ενσωματωμένη συνάρτηση `next()`) επιστρέφουν διαδοχικά στοιχεία στην ροή. Όταν όχι περισσότερα δεδομένα είναι διαθέσιμα εγείρεται μια εξαίρεση `StopIteration`. Σε αυτό το σημείο, το αντικείμενο iterator εξαντλείται και τυχόν περαιτέρω κλήσεις στη μέθοδο `__next__()` απλώς απλά εγείρουν ξανά το `StopIteration`. Οι iterators πρέπει να έχουν μια μέθοδο `__iter__()` που επιστρέφει το ίδιο το αντικείμενο iterator, έτσι ώστε κάθε iterator να είναι επίσης iterable και μπορεί να χρησιμοποιηθεί στα περισσότερα μέρη όπου γίνονται αποδεκτοί και άλλοι iterators. Μια αξιοσημείωτη εξαίρεση είναι ο κώδικας που επιχειρεί πολλαπλά περάσματα iteration. Ένα αντικείμενο κοντέινερ (όπως ένα `list`) παράγει έναν καθαρά νέο iterator κάθε φορά που κάθε φορά που μεταβιβάζεται στην συνάρτηση `iter()` ή τον χρησιμοποιείται σε έναν `for` βρόχο. Εάν επιχειρήσετε αυτό με έναν iterator απλώς θα επιστρέψετε το ίδιο εξαντλημένο αντικείμενο iterator που χρησιμοποιήθηκε στο προηγούμενο πέρασμα iteration, κάνοντας το να φαίνεται σαν ένα άδειο κοντέινερ.

Περισσότερες πληροφορίες μπορούν να βρεθούν στο `typeiter`.

Το CPython δεν εφαρμόζει με συνέπεια την απαίτηση να ορίζει ένας iterator `__iter__()`. Επίσης σημειώστε ότι η έκδοση CPython με ελεύθερη υποστήριξη νημάτων δεν εγγυάται την ασφάλεια νημάτων για διαδικασίες με iterators.

### 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()`.

Υπάρχουν διάφοροι τρόποι για να δημιουργήσετε μια συνάρτηση κλειδιού. Για παράδειγμα, η μέθοδος `str.lower()` μπορεί να χρησιμεύσει ως συνάρτηση κλειδί για την περίπτωση μη διάκρισης πεζών-κεφαλαίων. Εναλλακτικά, μια συνάρτηση κλειδιού μπορεί να δημιουργηθεί από μια *lambda* έκφραση όπως `lambda r: (r[0], r[2])`. Επίσης, `operator.attrgetter()`, `operator.itemgetter()` και `operator.methodcaller()` είναι τρεις κατασκευαστές βασικών συναρτήσεων. Βλ. το Ταξινόμηση HOW TO για παραδείγματα δημιουργίας και χρήσης βασικών συναρτήσεων.

### 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.

### λεξικός αναλυτής

Επίσημη ονομασία για τον *tokenizer* · βλ. *token*.

### λίστα

Ένα ενσωματωμένο Python *sequence*. Παρά το όνομα του, μοιάζει περισσότερο με έναν πίνακα σε άλλες γλώσσες παρά με μια συνδεδεμένη λίστα, καθώς η πρόσβαση στα στοιχεία είναι  $O(1)$ .

### 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

Ένα αντικείμενο που φορτώνει ένα module. Πρέπει να ορίζει τις μεθόδους `exec_module()` και `create_module()` για την υλοποίηση της διεπαφής *Loader*. Ένας loader συνήθως επιστρέφεται με ένα *finder*. Δείτε επίσης:

- *Finders and loaders*
- `importlib.abc.Loader`
- **PEP 302**

### τοπική κωδικοποίηση

Στο Unix, είναι η κωδικοποίηση της τοπική ρύθμισης `LC_CTYPE`. Μπορεί να ρυθμιστεί με `locale.setlocale(locale.LC_CTYPE, new_locale)`.

Στα Windows, είναι η code page ANSI (π.χ. "cp1252").

Στο Android και το VxWorks, η Python χρησιμοποιεί το "utf-8" ως τοπική κωδικοποίηση.

`locale.getencoding()` μπορεί να χρησιμοποιηθεί για την ανάκτηση της τοπικής κωδικοποίησης.

Βλ. επίσης το *filesystem encoding and error handler*.

### μαγική μέθοδος

Ένα άτυπο συνώνυμο για *special method*.

### mapping

Ένα αντικείμενο κοντέινερ που υποστηρίζει αυθαίρετες αναζητήσεις κλειδιών και υλοποιεί τις μεθόδους που καθορίζονται στο `collections.abc.Mapping` ή `collections.abc.MutableMapping` abstract base classes. Τα παραδείγματα περιλαμβάνουν `dict`, `collections.defaultdict`, `collections.OrderedDict` και `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

Η Σειρά Ανάλυσης Μεθόδων είναι η σειρά με την οποία οι βασικές κλάσεις αναζητούνται για ένα μέλος κατά την αναζήτηση. Βλ. `python_2.3_mro` για λεπτομέρειες του αλγορίθμου που χρησιμοποιείται από τον διερμηνέα της Python από την έκδοση 2.3.

### 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`.

Βλ. επίσης *Module specs*.

### 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
```

Ορισμένες αναγνωρισμένες πλειάδες είναι ενσωματωμένοι τύποι (όπως τα παραπάνω παραδείγματα). Εναλλακτικά, μια αναγνωρισμένη πλειάδα μπορεί να δημιουργηθεί από έναν ορισμό κανονικής κλάσης που κληρονομεί από `tuple` και που ορίζει έγκυρα πεδία. Μια τέτοια κλάση μπορεί να είναι γραμμένη με το χέρι ή μπορεί να δημιουργηθεί κληρονομώντας το `typing.NamedTuple`, ή με την factory συνάρτηση `collections.namedtuple()`. Οι τελευταίες τεχνικές προσθέτουν επίσης μερικές επιπλέον μεθόδους που μπορεί να μην βρεθούν σε χειρόγραφες ή ενσωματωμένες πλειάδες με όνομα.

### 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

Ένα *package* που χρησιμεύει μόνο ως κοντέινερ για υποπακέτα. Τα πακέτα χώρου ονομάτων μπορεί να μην έχουν φυσική αναπαράσταση και συγκεκριμένα να μην είναι σαν ένα *regular package* επειδή δεν έχουν το `__init__.py` αρχείο.

Τα πακέτα χώρου ονομάτων επιτρέπουν σε πολλά πακέτα με δυνατότητα εγκατάστασης μεμονωμένα να έχουν ένα κοινό γονικό πακέτο. Διαφορετικά, συνίσταται η χρήση ενός *regular package*.

Για περισσότερες πληροφορίες, δείτε το **PEP 420** και το *Namespace packages*.

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

Το παλιό όνομα για το είδος των κλάσεων χρησιμοποιείται πλέον για όλα τα αντικείμενα. Σε παλιότερες εκδόσεις της Python, μόνο οι κλάσεις νέου στυλ μπορούσαν να χρησιμοποιήσουν τις νεότερες, ευέλικτες δυνατότητες της Python όπως `__slots__`, descriptors, ιδιότητες `__getattr__()`, μέθοδοι κλάσης, και στατικές μέθοδοι.

### αντικείμενο

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

### βελτιστοποιημένο πεδίο ορατότητας (scope)

Ένα πεδίο ορατότητας (scope) όπου τα ονόματα των τοπικών μεταβλητών είναι γνωστό με βεβαιότητα στον μεταγλωττιστή κατά τη μεταγλώττιση του κώδικα, επιτρέποντας τη βελτιστοποίηση της πρόσβασης για ανάγνωση και εγγραφή σε αυτά τα ονόματα. Οι τοπικοί χώροι ονομάτων για συναρτήσεις,

γεννήτριες, συναρτήσεις *coroutine*, συμπτύξεις (*comprehensions*) και εκφράσεις γεννητριών βελτιστοποιούνται με αυτόν τον τρόπο. Σημείωση: οι περισσότερες βελτιστοποιήσεις του διερμηνέα εφαρμόζονται σε όλα τα πεδία ορατότητας· μόνο εκείνες που βασίζονται σε γνωστό σύνολο τοπικών και μη τοπικών μεταβλητών περιορίζονται σε βελτιστοποιημένα πεδία ορατότητας.

### πακέτο

Ένα Python *module* που μπορεί να περιέχει *submodules* ή αναδρομικά, υποπακέτα. Τεχνικά, ένα πακέτο είναι μια λειτουργική μονάδα Python με ένα `__path__` χαρακτηριστικό.

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 τη 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**

Ένα καλούμενο στη λίστα `sys.path_hooks`, το οποίο επιστρέφει ένα *path entry finder* εάν ξέρει πως να βρίσκει module σε μια συγκεκριμένη *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](#).

**τιμήμα**

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):
...             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 αναφορές

Ο αριθμός των αναφορών σε ένα αντικείμενο. Όταν το πλήθος αναφορών ενός αντικειμένου πέσει στο μηδέν, κατανέμεται. Μερικά αντικείμενα είναι *immortal* και έχουν πλήθος αναφορών που δεν τροποποιούνται ποτέ και επομένως τα αντικείμενα δεν κατανέμονται ποτέ. Η καταμέτρηση αναφορών γενικά δεν είναι ορατή στον κώδικα της Python, αλλά είναι βασικό στοιχείο της υλοποίησης *CPython*. Οι προγραμματιστές μπορούν να καλέσουν τη συνάρτηση `sys.getrefcount()` για να επιστρέψουν το πλήθος αναφορές για ένα συγκεκριμένο αντικείμενο.

In *CPython*, reference counts are not considered to be stable or well-defined values; the number of references to an object, and how that number is affected by Python code, may be different between versions.

### κανονικό πακέτο

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

Βλ. επίσης *namespace package*.

### REPL

Ακρωνύμιο του «read-eval-print loop», άλλη ονομασία για το *interactive* περιβάλλον του διερμηνέα.

### \_\_slots\_\_

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

### ακολουθία

Ένας *iterable* που υποστηρίζει την αποτελεσματική πρόσβαση στο στοιχείο χρησιμοποιώντας άκερτους δείκτες μέσω της ειδικής μεθόδου `__getitem__()` και ορίζει μια μέθοδο `__len__()` που

επιστρέφει το μήκος της ακολουθίας. Ορισμένοι ενσωματωμένοι τύποι ακολουθιών είναι `list`, `str`, `tuple`, και `bytes`. Σημειώστε ότι το `dict` υποστηρίζει επίσης `__getitem__()` και `__len__()`, αλλά θεωρείται αντιστοίχιση και όχι ακολουθία επειδή οι αναζητήσεις χρησιμοποιούν αυθαίρετα *hashable* κλειδιά παρά ακέραιοι.

Η αφηρημένη βασική κλάση `collections.abc.Sequence` ορίζει μια πολύ πιο πλούσια διεπαφή που ξεπερνά τα απλά `__getitem__()` και `__len__()`, `adding count()`, `index()`, `__contains__()`, και `__reversed__()`. Οι τύποι που υλοποιούν αυτήν την διευρυμένη διεπαφή μπορούν να καταχωρηθούν ρητά χρησιμοποιώντας `register()`. Για περισσότερη τεκμηρίωση σχετικά με τις μεθόδους ακολουθίας γενικά, ανατρέξτε στο *Common Sequence Operations*.

### 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.

### απαρχαιωμένη με ήπιο τρόπο

Ένα απαρχαιωμένο με ήπιο τρόπο API δεν θα πρέπει να χρησιμοποιείται σε νέο κώδικα, αλλά είναι ασφαλές σε ήδη υπάρχοντα κώδικα να το χρησιμοποιεί. Το API παραμένει τεκμηριωμένο και δοκιμασμένο, αλλά δεν θα ενισχυθεί περαιτέρω.

Η κατάργηση με ήπιο τρόπο, σε αντίθεση με την κανονική κατάργηση, δεν σχεδιάζει την κατάργηση του API και δεν θα εκπέμπει ειδοποιήσεις

Δείτε *PEP 387: Soft Deprecation*.

### ειδική μέθοδος

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

### standard library

The collection of *packages*, *modules* and *extension modules* distributed as a part of the official Python interpreter package. The exact membership of the collection may vary based on platform, available system libraries, or other criteria. Documentation can be found at *library-index*.

See also `sys.stdlib_module_names` for a list of all possible standard library module 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*.

### ελεγκτής στατικού τύπου

Ένα εξωτερικό εργαλείο όπου διαβάζει τον Python κώδικα και τον αναλύει, αναζητώντας προβλήματα όπως λανθασμένοι τύποι. Βλ. επίσης *type hints* και το *module typing*.

### stdlib

An abbreviation of *standard library*.

### strong reference

Στο C API της Python, μια ισχυρή αναφορά είναι μια αναφορά σε ένα αντικείμενο που ανήκει στον κώδικα που περιέχει την αναφορά. Η ισχυρή αναφορά λαμβάνεται καλώντας το `Py_INCREF()` όταν η αναφορά δημιουργείται και απελευθερώνεται με `Py_DECREF()` όταν διαγραφεί η αναφορά.

Η συνάρτηση `Py_NewRef()` μπορεί να χρησιμοποιηθεί για τη δημιουργία ισχυρής αναφοράς σε ένα αντικείμενο. Συνήθως, η συνάρτηση `Py_DECREF()` πρέπει να καλείται στην ισχυρή αναφορά πριν βγει από το εύρος της ισχυρής αναφοράς, για να αποφευχθεί η διαρροή μιας αναφοράς.

Βλ. επίσης *borrowed reference*.

### t-string

#### t-strings

String literals prefixed with `t` or `T` are commonly called «t-strings» which is short for *template string literals*.

### 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*.

### κατάσταση νήματος

Οι πληροφορίες που χρησιμοποιούνται από τη ροή εκτέλεσης της *CPython* για την εκτέλεση σε ένα νήμα λειτουργικού συστήματος. Για παράδειγμα, αυτό περιλαμβάνει την τρέχουσα εξαίρεση, εάν υπάρχει, και την κατάσταση του διερμηνέα bytecode.

Κάθε κατάσταση νήματος είναι συνδεδεμένη με ένα μόνο νήμα λειτουργικού συστήματος, αλλά τα νήματα μπορεί να έχουν πολλές διαθέσιμες καταστάσεις νήματος. Το πολύ, μία από αυτές μπορεί να είναι *attached* ταυτόχρονα.

Απαιτείται ένα *attached thread state* για την κλήση του μεγαλύτερου μέρους του C API της Python, εκτός εάν μια συνάρτηση τεκμηριώνεται ρητά από το αντίθετο. Ο διερμηνέας bytecode εκτελείται μόνο υπό κατάσταση συνημμένου νήματος.

Κάθε κατάσταση νήματος ανήκει σε έναν μόνο διερμηνέα, αλλά κάθε διερμηνέα μπορεί να έχει πολλές καταστάσεις νήματος, συμπεριλαμβανομένων πολλαπλών για το ίδιο νήμα λειτουργικού συστήματος. Οι καταστάσεις νήματος από πολλαπλούς διερμηνείς μπορεί να είναι συνδεδεμένες με το ίδιο νήμα, αλλά μόνο μία μπορεί να είναι *attached* σε αυτό το νήμα σε οποιαδήποτε δεδομένη στιγμή.

Δείτε το Thread State and the Global Interpreter Lock για περισσότερες πληροφορίες.

### λεκτικό σύμβολο (token)

Μια μικρή μονάδα πηγαιού κώδικα, που παράγεται από τον *lexical analyzer* (γνωστό και ως *αναλυτή (tokenizer)*). Ονόματα, αριθμοί, συμβολοσειρές, τελεστές αλλαγές γραμμής και παρόμοια στοιχεία αναπαρίστανται ως λεκτικά σύμβολα (tokens).

Το module `tokenize` εκθέτει τον λεξικό αναλυτή της Python. Το module `token` περιέχει πληροφορίες για τους διάφορους τύπους λεκτικών συμβόλων (tokens).

### 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**

Ο τύπος ενός Python αντικειμένου καθορίζει τι είδους αντικείμενο είναι• κάθε αντικείμενο έχει έναν τύπο. Ο τύπος ενός αντικειμένου είναι προσβάσιμος ως το χαρακτηριστικό `__class__` ή μπορεί να ανακτηθεί με `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.

Οι υποδείξεις τύπων (type hints) είναι προαιρετικές και δεν επιβάλλονται από την Python, αλλά είναι χρήσιμες για *static type checkers*. Μπορούν επίσης να βοηθήσουν τους IDEs με τη συμπλήρωση και την αναδιαμόρφωση κώδικα.

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*.

Βλ. *function annotation*, **PEP 484** και **PEP 526**, που περιγράφουν αυτή τη λειτουργία. Δείτε επίσης *annotations-howto* για βέλτιστες πρακτικές σχετικά με την εργασία με σχολιασμούς.

**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`.

#### **walrus operator**

A light-hearted way to refer to the *assignment expression* operator `:` because it looks a bit like a walrus if you turn your head.

#### **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.



---

### Σχετικά με την τεκμηρίωση

---

Η τεκμηρίωση της Python έχει δημιουργηθεί από τα [reStructuredText](#) sources του [Sphinx](#), έναν επεξεργαστή εγγράφων που έχει δημιουργηθεί ειδικά για τα έγγραφα της Python.

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., τον δημιουργό των αρχικών εργαλείων της τεκμηρίωσης της Python και συντάκτη αρκετού περιεχομένου·
- το [Docutils](#) πρότζεκτ για την δημιουργία των εφαρμογών reStructuredText και Docutils·
- Fredrik Lundh για το δικό του Alternative Python Reference πρότζεκτ από το οποίο το Sphinx πήρε πολύ καλές ιδέες.

### Β΄.1 Συντελεστές στη τεκμηρίωση της Python

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!





## Γ'.1 Η ιστορία του λογισμικού

Η Python δημιουργήθηκε στις αρχές του 1990 από τον Guido van Rossum στο Stichting Mathematisch Centrum (CWI, βλ. <https://www.cwi.nl>) στην Ολλανδία ως διάδοχος μια γλώσσας που ονομάζεται ABC. Ο Guido παραμένει ο κύριος συγγραφέας της Python, παρόλα αυτά περιλαμβάνει συνεισφορές και από άλλα άτομα.

Το 1995, ο Guido συνέχισε το έργο του για την Python στο Corporation for National Research Initiatives (CNRI, βλ. <https://www.cnri.reston.va.us>) στο Reston της Virginia, όπου κυκλοφόρησε πολλές εκδόσεις του λογισμικού.

Τον Μάιο του 2000, ο Guido και η βασική ομάδα ανάπτυξης της Python μετακόμισαν στο BeOpen.com για να σχηματίσουν την ομάδα BeOpen PythonLabs. Τον Οκτώβριο του ίδιου έτους, η ομάδα του PythonLabs μετακόμισε στην Digital Creations, η οποία μετατράπηκε σε Zope Corporation. Το 2001, το Python Software Foundation (PSF, βλ. <https://www.python.org/psf>) δημιουργήθηκε ως ένας μη κερδοσκοπικός οργανισμός με στόχο να κατέχει την πνευματική ιδιοκτησία που σχετίζεται με την Python. Η Zope Corporation ήταν μέλος-χορηγός του PSF.

Όλες οι εκδόσεις της Python είναι Ανοιχτού Κώδικα (βλ. <https://opensource.org> για τον ορισμό του Ανοιχτού Κώδικα). Ιστορικά οι περισσότερες, αλλά όχι όλες, εκδόσεις της Python ήταν επίσης συμβατές με την άδεια GPL: ο παρακάτω πίνακας συνοψίζει τις διάφορες εκδόσεις.

Έκδοση	Προερχόμενη από	Έτος	Ιδιοκτησία	Συμβατότητα GPL; (1)
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	ναι (2)
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

**i Σημείωση**

- (1) Η συμβατότητα με GPL δεν σημαίνει ότι διανέμεται η Python κάτω από την άδεια GPL. Όλες οι άδειες της Python, σε αντίθεση με την GPL, σας επιτρέπουν να διανείμετε μια τροποποιημένη έκδοση χωρίς να κάνετε τις αλλαγές σας, ανοιχτού κώδικα. Οι άδειες με συμβατότητα GPL καθιστούν δυνατό τον συνδυασμό της Python με άλλο λογισμικό που κυκλοφορεί με βάση της GPL, ενώ οι άλλες όχι.
- (2) Σύμφωνα με τον Richard Stallman, 1.6.1 δεν είναι συμβατή με την GPL, επειδή η άδεια της έχει νομική ρήτρα επιλογής, Σύμφωνα με το CNRI, ωστόσο, ο δικηγόρος του Stallman είπε στον δικηγόρο της CNRI ότι η 1.6.1 «δεν είναι συμβατή» με την GPL.

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

Το λογισμικό της Python και η τεκμηρίωση αδειοδοτούνται σύμφωνα με την άδεια χρήσης Python Software Foundation Έκδοση 2.

Ξεκινώντας από την Python 3.8.6, παραδείγματα, συνταγές και ο άλλος κώδικας στην τεκμηρίωση έχουν διπλή άδεια χρήσης, σύμφωνα με την Άδεια PSF Έκδοση 2 και της *Zero-Clause BSD άδεια*.

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 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using this software ("Python") in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights Reserved" are retained in Python alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made.

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

`→to Python.`

4. PSF is making Python available to Licensee on an "AS IS" basis.  
`PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY`  
`→OF`  
`EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY`  
`→REPRESENTATION OR`  
`WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR`  
`→THAT THE`  
`USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.`
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON  
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A  
`→RESULT OF`  
MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE  
THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material  
`→breach of`  
its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any  
`→relationship`  
of agency, partnership, or joint venture between PSF and Licensee. `→`  
`→This License`  
Agreement does not grant permission to use PSF trademarks or trade name  
`→in a`  
trademark sense to endorse or promote products or services of Licensee,  
`→or any`  
third party.
8. By copying, installing or otherwise using Python, Licensee agrees  
to be bound by the terms and conditions of this License Agreement.

## 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`

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
Software, alone or in any derivative version prepared by Licensee.

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

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

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

6. This License Agreement shall be governed by and interpreted in all_
→respects
   by the law of the State of California, excluding conflict of law_
→provisions.
   Nothing in this License Agreement shall be deemed to create any_
→relationship of
   agency, partnership, or joint venture between BeOpen and Licensee. _
→This License
   Agreement does not grant permission to use BeOpen trademarks or trade_
→names in a
   trademark sense to endorse or promote products or services of Licensee,_
→or any
   third party. As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the_
→permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees_
→to be
   bound by the terms and conditions of this License Agreement.
```

## 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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

grants Licensee a nonexclusive, royalty-free, world-wide license to  
 →reproduce,  
 analyze, test, perform and/or display publicly, prepare derivative  
 →works,  
 distribute, and otherwise use Python 1.6.1 alone or in any derivative  
 →version,  
 provided, however, that CNRI's License Agreement and CNRI's notice of  
 →copyright,  
 i.e., "Copyright © 1995-2001 Corporation for National Research  
 →Initiatives; All  
 Rights Reserved" are retained in Python 1.6.1 alone or in any  
 →derivative version  
 prepared by Licensee. Alternately, in lieu of CNRI's License Agreement,  
 Licensee may substitute the following text (omitting the quotes):  
 →"Python 1.6.1  
 is made available subject to the terms and conditions in CNRI's License  
 Agreement. This Agreement together with Python 1.6.1 may be located on  
 →the  
 internet using the following unique, persistent identifier (known as a  
 →handle):  
 1895.22/1013. This Agreement may also be obtained from a proxy server  
 →on the  
 internet using the following URL: <http://hdl.handle.net/1895.22/1013>".

3. In the event Licensee prepares a derivative work that is based on or  
 incorporates Python 1.6.1 or any part thereof, and wants to make the  
 →derivative  
 work available to others as provided herein, then Licensee hereby  
 →agrees to  
 include in any such work a brief summary of the changes made to Python  
 →1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.   
 →CNRI  
 MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF  
 →EXAMPLE,  
 BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR  
 →WARRANTY  
 OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE  
 →USE OF  
 PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1  
 →FOR  
 ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF  
 MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY  
 →DERIVATIVE  
 THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

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

7. This License Agreement shall be governed by the federal intellectual  
 →property  
 law of the United States, including without limitation the federal

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
→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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

→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 ΑΔΕΙΑ ΓΙΑ ΤΟΝ ΚΩΔΙΚΑ ΣΤΗΝ ΤΕΚΜΗΡΙΩΣΗ ΤΗΣ PYTHON

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

Η επέκταση `_random` C που βρίσκεται κάτω από την ενότητα `random` περιλαμβάνει έναν κώδικα με βάση μια λήψη από <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. Ακολουθούν κατά λέξη τα σχόλια από το αρχικό κώδικα:

```

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.

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

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

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT\_

→OWNER OR

CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

## Γ'.3.2 Sockets

Η ενότητα socket χρησιμοποιεί τις συναρτήσεις, `getaddrinfo()`, και `getnameinfo()`, τα οποία έχουν υλοποιηθεί σε διαφορετικά αρχεία από το WIDE Έργο, <https://www.wide.ad.jp/>.

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

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

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

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



### Γ'.3.3 Ασύγχρονες socket υπηρεσίες

Οι ενότητες `test.support.asyncio` και `test.support.asyncore` περιέχουν την παρακάτω ειδοποίηση:

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:

portions copyright 2001, Autonomous Zones Industries, Inc., all rights...  
err... reserved and offered to the public under the terms of the

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
Python 2.2 license.  
Author: Zooko O'Whielacronx  
http://zooko.com/  
mailto:zooko@zooko.com
```

```
Copyright 2000, Mojam Media, Inc., all rights reserved.  
Author: Skip Montanaro
```

```
Copyright 1999, Bioreason, Inc., all rights reserved.  
Author: Andrew Dalke
```

```
Copyright 1995-1997, Automatrix, Inc., all rights reserved.  
Author: Skip Montanaro
```

```
Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.
```

```
Permission to use, copy, modify, and distribute this Python software and  
its associated documentation for any purpose without fee is hereby  
granted, provided that the above copyright notice appears in all copies,  
and that both that copyright notice and this permission notice appear in  
supporting documentation, and that the name of neither Automatrix,  
Bioreason or Mojam Media be used in advertising or publicity pertaining to  
distribution of the software without specific, written prior permission.
```

### Γ'.3.6 Συναρτήσεις UUencode και UUdecode

Ο `uu` κωδικοποιητής περιέχει την παρακάτω ειδοποίηση:

```
Copyright 1994 by Lance Ellinghouse  
Cathedral City, California Republic, United States of America.  
All Rights Reserved  
Permission to use, copy, modify, and distribute this software and its  
documentation for any purpose and without fee is hereby granted,  
provided that the above copyright notice appear in all copies and that  
both that copyright notice and this permission notice appear in  
supporting documentation, and that the name of Lance Ellinghouse  
not be used in advertising or publicity pertaining to distribution  
of the software without specific, written prior permission.  
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO  
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND  
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE  
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES  
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN  
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT  
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.  
  
Modified by Jack Jansen, CWI, July 1995:  
- Use binascii module to do the actual line-by-line conversion  
  between ascii and binary. This results in a 1000-fold speedup. The C  
  version is still 5 times faster, though.  
- Arguments more compliant with Python standard
```

### 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 MERCHANTABILITY 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`

Η ενότητα `test.test_epoll` περιέχει την παρακάτω ειδοποίηση:

Copyright (c) 2001–2006 Twisted Matrix Laboratories.

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

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

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

### 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 του 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>
```

```
Original location:
  https://github.com/majek/csiphash/
```

```
Solution inspired by code from:
  Samuel Neves (supercop/crypto_auth/siphhash24/little)
  djb (supercop/crypto_auth/siphhash24/little2)
  Jean-Philippe Aumasson (https://131002.net/siphhash/siphhash24.c)
```

### Γ'.3.11 strtod και dtoa

Το αρχείο `Python/dtoa.c`, που παρέχει τις συναρτήσεις `dtoa` και `strtod` της C για μετατροπή των C doubles προς και από strings, προέρχεται από το ομώνυμο αρχείο του David M. Gay, προς το παρόν διαθέσιμο από <https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c>. Το αρχικό αρχείο, όπως ανακτήθηκε στις 16 Μαρτίου, 2009, περιέχει τα ακόλουθα πνευματικά δικαιώματα και την ειδοποίηση αδειοδότησης:

```

/*****
 *
 * 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

Οι μονάδες `hashlib`, `posix` και `ssl` χρησιμοποιούν την βιβλιοθήκη OpenSSL για επιπλέον απόδοση, εάν διατίθενται από το λειτουργικό σύστημα. Επιπλέον, τα προγράμματα εγκατάστασης για την Python για Windows και macOS, ενδέχεται να περιλαμβάνουν ένα αντίγραφο των βιβλιοθηκών OpenSSL, επομένως συμπεριλαμβάνουμε ένα αντίγραφο της άδειας OpenSSL εδώ. Για την έκδοση OpenSSL 3.0 και για νεότερες εκδόσεις που προέρχονται από αυτή, ισχύει η άδεια Apache v2:

```

                                Apache License
                                Version 2.0, January 2004
                                https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

"License" shall mean the terms and conditions for use, reproduction,
and distribution as defined by Sections 1 through 9 of this document.

"Licenser" shall mean the copyright owner or entity authorized by
the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all
other entities that control, are controlled by, or are under common
control with that entity. For the purposes of this definition,
"control" means (i) the power, direct or indirect, to cause the
direction or management of such entity, whether by contract or
otherwise, or (ii) ownership of fifty percent (50%) or more of the
outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
  - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
  - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
  - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
  - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

### Γ'.3.13 expat

Η επέκταση pyexpat δημιουργείται χρησιμοποιώντας ένα συμπεριλαμβανόμενο αντίγραφο των πηγών expat, εκτός εάν η έκδοση έχει την ρύθμιση --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

(συνέχεια στην επόμενη σελίδα)



(συνεχίζεται από την προηγούμενη σελίδα)

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

Η επέκταση της C `_ctypes` που βρίσκεται κάτω από την ενότητα `ctypes` δημιουργείται χρησιμοποιώντας ένα συμπεριλαμβανόμενο αντίγραφο των πηγών `libffi`, εκτός εάν η έκδοση έχει την ρύθμιση `--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:

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

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 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

Η επέκταση `_decimal` που βρίσκεται κάτω από την ενότητα `decimal` είναι φτιαγμένη χρησιμοποιώντας ένα συμπεριλαμβανόμενο αντίγραφο της βιβλιοθήκης `libmpdec`, εκτός αν η έκδοση έχει ρύθμιση `--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:

- \* 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,

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

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'.3.19 mimalloc

MIT Άδεια:

Copyright (c) 2018–2021 Microsoft Corporation, Daan Leijen

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

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

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

### Γ'.3.20 asyncio

Μέρη της ενότητας asyncio ενσωματώνονται από το [uvloop 0.16](#), η οποία διανέμεται με άδεια MIT:

Copyright (c) 2015–2021 MagicStack Inc. <http://magic.io>

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

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

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

(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'.3.21 Καθολικές Απεριόριστες Ακολουθίες (KAA)

Το αρχείο `Python/qsbr.c` είναι προσαρμοσμένο από το σύστημα ασφαλούς ανάκτησης μνήμης «Global Unbounded Sequences» του FreeBSD, που υλοποιείται στο `subr_smr.c`. Το αρχείο διανέμεται υπό την Άδεια 2-Clause BSD:

```

Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

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

```

### Γ'.3.22 Δεσμεύσεις Zstandard

Οι δεσμεύσεις Zstandard στα `Modules/_zstd` και `Lib/compression/zstd` βασίζονται σε κώδικα από τη βιβλιοθήκη `pyzstd library`, πνευματικής ιδιοκτησίας του Ma Lin και των συνεργατών του. Ο κώδικας της `pyzstd` διανέμεται υπό την άδεια 3-Clause BSD.

```

Copyright (c) 2020-present, Ma Lin and contributors.
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.

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

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

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

→ARE

DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER 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.

---

### Copyright

---

Η Python και αυτή η τεκμηρίωση είναι:

Copyright © 2001 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. Όλα τα δικαιώματα διατηρούνται.

Copyright © 1995-2000 Corporation for National Research Initiatives. Όλα τα δικαιώματα διατηρούνται.

Copyright © 1991-1995 Stichting Mathematisch Centrum. Όλα τα δικαιώματα διατηρούνται.

---

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





## μη-αλφαβητικά

- `...`, 161
  - ellipsis literal, 24
- `'''`
  - string literal, 12
- `.` (*dot*)
  - attribute reference, 94
  - in numeric literal, 19
- `!` (*exclamation*)
  - in formatted string literal, 16
- `-` (*minus*)
  - binary operator, 98
  - unary operator, 97
- `;` (*semicolon*), 117
- `'` (*single quote*)
  - string literal, 12
- `!` patterns, 125
- `"` (*double quote*)
  - string literal, 12
- `"""`
  - string literal, 12
- `#` (*hash*)
  - comment, 8
  - source encoding declaration, 8
- `%` (*percent*)
  - operator, 98
- `%=`
  - augmented assignment, 107
- `&` (*ampersand*)
  - operator, 99
- `&=`
  - augmented assignment, 107
- `()` (*parentheses*)
  - call, 95
  - class definition, 133
  - function definition, 131
  - generator expression, 89
  - in assignment target list, 106
  - tuple display, 87
- `*` (*asterisk*)
  - function definition, 132
  - import statement, 113
  - in assignment target list, 106
  - in expression lists, 103
  - in function calls, 96
  - operator, 98
- `**`
  - function definition, 132
  - in dictionary displays, 89
  - in function calls, 96
  - operator, 97
- `**=`
  - augmented assignment, 107
- `*=`
  - augmented assignment, 107
- `+` (*plus*)
  - binary operator, 98
  - unary operator, 97
- `+=`
  - augmented assignment, 107
- `,` (*comma*), 87
  - argument list, 95
  - expression list, 88, 103, 109, 133
  - identifier list, 115
  - import statement, 112
  - in dictionary displays, 88
  - in target list, 106
  - parameter list, 131
  - slicing, 94
  - with statement, 122
- `/` (*slash*)
  - function definition, 132
  - operator, 98
- `//`
  - operator, 98
- `//=`
  - augmented assignment, 107
- `/=`
  - augmented assignment, 107
- `0b`
  - integer literal, 19
- `0o`
  - integer literal, 19
- `0x`
  - integer literal, 19
- `:` (*colon*)
  - annotated variable, 108

compound statement, 118, 119, 122, 123, 131, 133  
function annotations, 132  
in dictionary expressions, 88  
in formatted string literal, 16  
lambda expression, 103  
slicing, 94  
:= (*colon equals*), 102  
< (*less*)  
operator, 99  
<<  
operator, 99  
<<=  
augmented assignment, 107  
<=  
operator, 99  
!=  
operator, 99  
-=  
augmented assignment, 107  
= (*equals*)  
assignment statement, 106  
class definition, 50  
for help in debugging using string literals, 16  
function definition, 132  
in function calls, 95  
==  
operator, 99  
->  
function annotations, 132  
> (*greater*)  
operator, 99  
>=  
operator, 99  
>>  
operator, 99  
>>=  
augmented assignment, 107  
>>>, 161  
@ (*at*)  
class definition, 133  
function definition, 131  
operator, 98  
AS pattern, OR pattern, capture pattern, wildcard pattern, 125  
ASCII, 12  
AssertionError  
exception, 109  
AttributeError  
exception, 94  
BDFL, 163  
BNF, 4, 85  
Boolean  
object, 25  
operation, 102  
C, 13  
language, 24, 25, 30, 99  
CPython, 165  
C-contiguous, 165  
Conditional  
expression, 102  
DEDENT token, 9, 118  
EAFP, 167  
Ellipsis  
object, 24  
False, 25  
Fortran contiguous, 165  
GIL, 169  
GeneratorExit  
exception, 91, 93  
IDLE, 170  
INDENT token, 9  
ImportError  
exception, 112  
Java  
language, 25  
LBYL, 172  
MRO, 173  
NEWLINE token, 7, 118  
NameError  
exception, 85  
NameError (*ενοσωματωμένη εξαίρεση*), 68  
None  
object, 24, 105  
NotImplemented  
object, 24  
PEP, 176  
PYTHONHASHSEED, 45  
PYTHONNODEBUGRANGES, 38  
PYTHONPATH, 80  
PYTHON\_GIL, 169  
Python 3000, 176  
Python Enhancement Proposals  
PEP 1, 176  
PEP 8, 100  
PEP 236, 114  
PEP 238, 168  
PEP 252, 47  
PEP 255, 90  
PEP 278, 180  
PEP 302, 73, 83, 172  
PEP 308, 103  
PEP 318, 133, 134  
PEP 328, 83  
PEP 338, 83  
PEP 342, 90  
PEP 343, 60, 123, 165  
PEP 362, 162, 175  
PEP 366, 31, 83  
PEP 380, 90  
PEP 411, 176  
PEP 414, 13  
PEP 420, 73, 74, 79, 83, 174, 176  
PEP 443, 169  
PEP 448, 89, 96, 103

- PEP 451, 83
- PEP 483, 169
- PEP 484, 53, 108, 133, 161, 168, 169, 180
- PEP 492, 63, 90, 136, 162, 163, 165
- PEP 498, 18, 167
- PEP 519, 176
- PEP 525, 90, 162
- PEP 526, 108, 133, 161, 180
- PEP 530, 88
- PEP 560, 51, 55
- PEP 562, 47
- PEP 563, 114, 133
- PEP 570, 132
- PEP 572, 89, 103, 127
- PEP 585, 169
- PEP 614, 132, 134
- PEP 617, 5
- PEP 626, 39
- PEP 634, 60, 124, 131
- PEP 636, 124, 131
- PEP 646, 94, 104, 132
- PEP 649, 28, 33, 34, 61, 70, 161
- PEP 683, 170
- PEP 688, 61
- PEP 695, 70, 116
- PEP 696, 70, 136
- PEP 703, 168, 170
- PEP 749, 70, 140
- PEP 749#pep749-metaclasses, 34
- PEP 758, 119
- PEP 765, 121
- PEP 3104, 115
- PEP 3107, 133
- PEP 3115, 52, 134
- PEP 3116, 180
- PEP 3119, 53
- PEP 3120, 7
- PEP 3129, 133, 134
- PEP 3131, 11
- PEP 3132, 107
- PEP 3135, 53
- PEP 3147, 32
- PEP 3155, 177
- Pythonic, 176
- REPL, 177
- Standard C, 13
- StopAsyncIteration
  - exception, 93
- StopIteration
  - exception, 91, 110
- SystemExit (ενσωματωμένη εξαίρεση), 71
- True, 25
- TypeError
  - exception, 97
- UNIX, 141
- UnboundLocalError, 68
- Unicode, 26
- ValueError
  - exception, 99
- Windows, 141
- Zen της Python, 181
- ZeroDivisionError
  - exception, 98
- [ ] (*square brackets*)
  - in assignment target list, 106
  - list expression, 88
  - subscription, 94
- \ (*backslash*)
  - escape sequence, 13
- \N
  - escape sequence, 13
- \U
  - escape sequence, 13
- \\
  - escape sequence, 13
- \a
  - escape sequence, 13
- \b
  - escape sequence, 13
- \f
  - escape sequence, 13
- \n
  - escape sequence, 13
- \r
  - escape sequence, 13
- \t
  - escape sequence, 13
- \u
  - escape sequence, 13
- \v
  - escape sequence, 13
- \x
  - escape sequence, 13
- ^ (*caret*)
  - operator, 99
- ^=
  - augmented assignment, 107
- \_ (*underscore*)
  - in numeric literal, 19
- \_, identifiers, 11
- \_\_, identifiers, 11
- \_\_abs\_\_() (μέθοδος της *object*), 59
- \_\_add\_\_() (μέθοδος της *object*), 57
- \_\_aenter\_\_() (μέθοδος της *object*), 64
- \_\_aexit\_\_() (μέθοδος της *object*), 64
- \_\_aiter\_\_() (μέθοδος της *object*), 64
- \_\_all\_\_ (optional module attribute), 113
- \_\_and\_\_() (μέθοδος της *object*), 57
- \_\_anext\_\_() (μέθοδος της *agen*), 93
- \_\_anext\_\_() (μέθοδος της *object*), 64
- \_\_annotate\_\_ (class attribute), 34
- \_\_annotate\_\_ (function attribute), 28
- \_\_annotate\_\_ (module attribute), 30
- \_\_annotate\_\_ (ιδιότητα της *function*), 28
- \_\_annotate\_\_ (ιδιότητα της *module*), 33
- \_\_annotate\_\_() (μέθοδος της *object*), 61

`__annotate__()` (μέθοδος της *type*), 34  
`__annotations__` (*class attribute*), 34  
`__annotations__` (*function attribute*), 28  
`__annotations__` (*module attribute*), 30  
`__annotations__` (ιδιότητα της *function*), 28  
`__annotations__` (ιδιότητα της *module*), 33  
`__annotations__` (ιδιότητα της *object*), 61  
`__annotations__` (ιδιότητα της *type*), 34  
`__await__()` (μέθοδος της *object*), 63  
`__bases__` (*class attribute*), 34  
`__bases__` (ιδιότητα της *type*), 34  
`__bool__()` (*object method*), 56  
`__bool__()` (μέθοδος της *object*), 45  
`__buffer__()` (μέθοδος της *object*), 60  
`__bytes__()` (μέθοδος της *object*), 43  
`__cached__` (*module attribute*), 30  
`__cached__` (ιδιότητα της *module*), 32  
`__call__()` (*object method*), 97  
`__call__()` (μέθοδος της *object*), 55  
`__cause__` (*exception attribute*), 111  
`__ceil__()` (μέθοδος της *object*), 59  
`__class__` (*instance attribute*), 35  
`__class__` (*method cell*), 52  
`__class__` (*module attribute*), 47  
`__class__` (ιδιότητα της *object*), 35  
`__class_getitem__()` (μέθοδος κλάσης της *object*), 53  
`__classcell__` (*class namespace entry*), 52  
`__closure__` (*function attribute*), 28  
`__closure__` (ιδιότητα της *function*), 28  
`__code__` (*function attribute*), 28  
`__code__` (ιδιότητα της *function*), 28  
`__complex__()` (μέθοδος της *object*), 59  
`__contains__()` (μέθοδος της *object*), 57  
`__context__` (*exception attribute*), 111  
`__debug__`, 109  
`__defaults__` (*function attribute*), 28  
`__defaults__` (ιδιότητα της *function*), 28  
`__del__()` (μέθοδος της *object*), 42  
`__delattr__()` (μέθοδος της *object*), 46  
`__delete__()` (μέθοδος της *object*), 48  
`__delitem__()` (μέθοδος της *object*), 57  
`__dict__` (*class attribute*), 34  
`__dict__` (*function attribute*), 28  
`__dict__` (*instance attribute*), 35  
`__dict__` (*module attribute*), 33  
`__dict__` (ιδιότητα της *function*), 28  
`__dict__` (ιδιότητα της *module*), 33  
`__dict__` (ιδιότητα της *object*), 35  
`__dict__` (ιδιότητα της *type*), 34  
`__dir__` (*module attribute*), 47  
`__dir__()` (μέθοδος της *object*), 46  
`__divmod__()` (μέθοδος της *object*), 57  
`__doc__` (*class attribute*), 34  
`__doc__` (*function attribute*), 28  
`__doc__` (*method attribute*), 29  
`__doc__` (*module attribute*), 30  
`__doc__` (ιδιότητα της *function*), 28  
`__doc__` (ιδιότητα της *method*), 29  
`__doc__` (ιδιότητα της *module*), 33  
`__doc__` (ιδιότητα της *type*), 34  
`__enter__()` (μέθοδος της *object*), 60  
`__eq__()` (μέθοδος της *object*), 44  
`__exit__()` (μέθοδος της *object*), 60  
`__file__` (*module attribute*), 30  
`__file__` (ιδιότητα της *module*), 32  
`__firstlineno__` (*class attribute*), 34  
`__firstlineno__` (ιδιότητα της *type*), 34  
`__float__()` (μέθοδος της *object*), 59  
`__floor__()` (μέθοδος της *object*), 59  
`__floordiv__()` (μέθοδος της *object*), 57  
`__format__()` (μέθοδος της *object*), 43  
`__func__` (*method attribute*), 29  
`__func__` (ιδιότητα της *method*), 29  
`__future__`, 168  
    *future statement*, 114  
`__ge__()` (μέθοδος της *object*), 44  
`__get__()` (μέθοδος της *object*), 47  
`__getattr__` (*module attribute*), 47  
`__getattr__()` (μέθοδος της *object*), 45  
`__getattribute__()` (μέθοδος της *object*), 46  
`__getitem__()` (*mapping object method*), 41  
`__getitem__()` (μέθοδος της *object*), 56  
`__globals__` (*function attribute*), 28  
`__globals__` (ιδιότητα της *function*), 28  
`__gt__()` (μέθοδος της *object*), 44  
`__hash__()` (μέθοδος της *object*), 44  
`__iadd__()` (μέθοδος της *object*), 58  
`__iand__()` (μέθοδος της *object*), 58  
`__ifloordiv__()` (μέθοδος της *object*), 58  
`__ilshift__()` (μέθοδος της *object*), 58  
`__imatmul__()` (μέθοδος της *object*), 58  
`__imod__()` (μέθοδος της *object*), 58  
`__imul__()` (μέθοδος της *object*), 58  
`__index__()` (μέθοδος της *object*), 59  
`__init__()` (μέθοδος της *object*), 42  
`__init_subclass__()` (μέθοδος κλάσης της *object*), 49  
`__instancecheck__()` (μέθοδος της *type*), 53  
`__int__()` (μέθοδος της *object*), 59  
`__invert__()` (μέθοδος της *object*), 59  
`__ior__()` (μέθοδος της *object*), 58  
`__ipow__()` (μέθοδος της *object*), 58  
`__irshift__()` (μέθοδος της *object*), 58  
`__isub__()` (μέθοδος της *object*), 58  
`__iter__()` (μέθοδος της *object*), 57  
`__itruediv__()` (μέθοδος της *object*), 58  
`__ixor__()` (μέθοδος της *object*), 58  
`__kwdefaults__` (*function attribute*), 28  
`__kwdefaults__` (ιδιότητα της *function*), 28  
`__le__()` (μέθοδος της *object*), 44  
`__len__()` (*mapping object method*), 45  
`__len__()` (μέθοδος της *object*), 56  
`__length_hint__()` (μέθοδος της *object*), 56  
`__loader__` (*module attribute*), 30  
`__loader__` (ιδιότητα της *module*), 32

- `__lshift__()` (μέθοδος της *object*), 57
- `__lt__()` (μέθοδος της *object*), 44
- `__main__`
  - module, 69, 141
- `__matmul__()` (μέθοδος της *object*), 57
- `__missing__()` (μέθοδος της *object*), 57
- `__mod__()` (μέθοδος της *object*), 57
- `__module__` (class attribute), 34
- `__module__` (function attribute), 28
- `__module__` (method attribute), 29
- `__module__` (ιδιότητα της function), 28
- `__module__` (ιδιότητα της method), 29
- `__module__` (ιδιότητα της type), 34
- `__mro__` (ιδιότητα της type), 34
- `__mro_entries__()` (μέθοδος της *object*), 51
- `__mul__()` (μέθοδος της *object*), 57
- `__name__` (class attribute), 34
- `__name__` (function attribute), 28
- `__name__` (method attribute), 29
- `__name__` (module attribute), 30
- `__name__` (ιδιότητα της function), 28
- `__name__` (ιδιότητα της method), 29
- `__name__` (ιδιότητα της module), 31
- `__name__` (ιδιότητα της type), 34
- `__ne__()` (μέθοδος της *object*), 44
- `__neg__()` (μέθοδος της *object*), 59
- `__new__()` (μέθοδος της *object*), 42
- `__next__()` (μέθοδος της *generator*), 91
- `__objclass__` (ιδιότητα της *object*), 48
- `__or__()` (μέθοδος της *object*), 57
- `__package__` (module attribute), 30
- `__package__` (ιδιότητα της module), 31
- `__path__` (module attribute), 30
- `__path__` (ιδιότητα της module), 32
- `__pos__()` (μέθοδος της *object*), 59
- `__pow__()` (μέθοδος της *object*), 57
- `__prepare__` (metaclass method), 52
- `__qualname__` (ιδιότητα της function), 28
- `__qualname__` (ιδιότητα της type), 34
- `__radd__()` (μέθοδος της *object*), 58
- `__rand__()` (μέθοδος της *object*), 58
- `__rdivmod__()` (μέθοδος της *object*), 58
- `__release_buffer__()` (μέθοδος της *object*), 61
- `__repr__()` (μέθοδος της *object*), 43
- `__reversed__()` (μέθοδος της *object*), 57
- `__rfloordiv__()` (μέθοδος της *object*), 58
- `__rlshift__()` (μέθοδος της *object*), 58
- `__rmatmul__()` (μέθοδος της *object*), 58
- `__rmod__()` (μέθοδος της *object*), 58
- `__rmul__()` (μέθοδος της *object*), 58
- `__ror__()` (μέθοδος της *object*), 58
- `__round__()` (μέθοδος της *object*), 59
- `__rpow__()` (μέθοδος της *object*), 58
- `__rrshift__()` (μέθοδος της *object*), 58
- `__rshift__()` (μέθοδος της *object*), 57
- `__rsub__()` (μέθοδος της *object*), 58
- `__rtruediv__()` (μέθοδος της *object*), 58
- `__rxor__()` (μέθοδος της *object*), 58
- `__self__` (method attribute), 29
- `__self__` (ιδιότητα της method), 29
- `__set__()` (μέθοδος της *object*), 48
- `__set_name__()` (μέθοδος της *object*), 50
- `__setattr__()` (μέθοδος της *object*), 46
- `__setitem__()` (μέθοδος της *object*), 56
- `__slots__`, 177
- `__spec__` (module attribute), 30
- `__spec__` (ιδιότητα της module), 31
- `__static_attributes__` (class attribute), 34
- `__static_attributes__` (ιδιότητα της type), 34
- `__str__()` (μέθοδος της *object*), 43
- `__sub__()` (μέθοδος της *object*), 57
- `__subclasscheck__()` (μέθοδος της type), 53
- `__subclasses__()` (μέθοδος της type), 35
- `__traceback__` (exception attribute), 110
- `__truediv__()` (μέθοδος της *object*), 57
- `__trunc__()` (μέθοδος της *object*), 59
- `__type_params__` (class attribute), 34
- `__type_params__` (function attribute), 28
- `__type_params__` (ιδιότητα της function), 28
- `__type_params__` (ιδιότητα της type), 34
- `__xor__()` (μέθοδος της *object*), 57
- abs
  - built-in function, 59
- aclose() (μέθοδος της *agen*), 93
- addition, 98
- and
  - bitwise, 99
  - operator, 102
- annotated
  - assignment, 108
- annotation, 161
- annotation μεταβλητής, 180
- annotations
  - function, 132
- anonymous
  - function, 103
- argument
  - call semantics, 95
  - function, 27
  - function definition, 132
- arithmetic
  - conversion, 85
  - operation, binary, 97
  - operation, unary, 97
- array
  - module, 26
- as
  - except clause, 119
  - import statement, 112
  - keyword, 112, 119, 122, 123
  - match statement, 123
  - with statement, 122
- asend() (μέθοδος της *agen*), 93
- assert
  - statement, 109
- assertions

- debugging, 109
- assignment
  - annotated, 108
  - attribute, 106
  - augmented, 107
  - class attribute, 33
  - class instance attribute, 35
  - expression, 102
  - slicing, 107
  - statement, 26, 106
  - subscription, 107
  - target list, 106
- assignment expression, 102
- async
  - keyword, 134
- async def
  - statement, 134
- async for
  - in comprehensions, 87
  - statement, 135
- async with
  - statement, 135
- asynchronous generator
  - asynchronous iterator, 30
  - function, 30
- asynchronous-generator
  - object, 93
- athrow() (μέθοδος της *agen*), 93
- atom, 85
- attribute, 24
  - assignment, 106
  - assignment, class, 33
  - assignment, class instance, 35
  - class, 33
  - class instance, 35
  - deletion, 109
  - generic special, 24
  - reference, 94
  - special, 24
- augmented
  - assignment, 107
- await
  - in comprehensions, 88
  - keyword, 97, 134
- awaitable, **163**
- b'
  - bytes literal, 15
- b"
  - bytes literal, 15
- backslash character, 8
- binary
  - arithmetic operation, 97
  - bitwise operation, 99
- binary literal, 18
- binding
  - global name, 115
  - name, 106, 112, 113, 131, 133
- bitwise
  - and, 99
  - operation, binary, 99
  - operation, unary, 97
  - or, 99
  - xor, 99
- blank line, 9
- break
  - statement, **112**, 118, 121
- built-in
  - method, 30
- built-in function
  - abs, 59
  - bytes, 43
  - call, 96
  - chr, 26
  - compile, 115
  - complex, 59
  - divmod, 58
  - eval, 115, 142
  - exec, 115
  - float, 59
  - hash, 44
  - id, 23
  - int, 59
  - len, 26, 27, 56
  - object, 30, 96
  - open, 35
  - ord, 26
  - pow, 58
  - print, 43
  - range, 119
  - repr, 105
  - round, 59
  - slice, 41
  - type, 23, 50
- built-in method
  - call, 96
  - object, 30, 96
- builtins
  - module, 141
- byte, 26
- bytearray, 26
- bytecode, 36, **164**
- bytes, 26
  - built-in function, 43
- bytes literal, 12
- bytes-like αντικείμενα, **163**
- call, 95
  - built-in function, 96
  - built-in method, 96
  - class instance, 96
  - class object, 33, 96
  - function, 27, 96
  - instance, 55, 97
  - method, 96
  - procedure, 105
  - user-defined function, 96
- callable, **164**



- object, 27, 95
- callback, **164**
- case
  - keyword, **123**
  - match, **123**
- case block, **125**
- chaining
  - comparisons, **99**
  - exception, **111**
- character, 26, 94
- chr
  - built-in function, 26
- class
  - attribute, 33
  - attribute assignment, 33
  - body, 52
  - constructor, 42
  - definition, 109, 133
  - instance, 35
  - name, 133
  - object, 33, 96, 133
  - statement, 133
- class instance
  - attribute, 35
  - attribute assignment, 35
  - call, 96
  - object, 33, 35, 96
- class object
  - call, 33, 96
- clause, **117**
- clear() (μέθοδος της frame), 40
- close() (μέθοδος της coroutine), 64
- close() (μέθοδος της generator), 91
- co\_argcount (code object attribute), 36
- co\_argcount (ιδιότητα της codeobject), 37
- co\_cellvars (code object attribute), 36
- co\_cellvars (ιδιότητα της codeobject), 37
- co\_code (code object attribute), 36
- co\_code (ιδιότητα της codeobject), 37
- co\_consts (code object attribute), 36
- co\_consts (ιδιότητα της codeobject), 37
- co\_filename (code object attribute), 36
- co\_filename (ιδιότητα της codeobject), 37
- co\_firstlineno (code object attribute), 36
- co\_firstlineno (ιδιότητα της codeobject), 37
- co\_flags (code object attribute), 36
- co\_flags (ιδιότητα της codeobject), 37
- co\_freevars (code object attribute), 36
- co\_freevars (ιδιότητα της codeobject), 37
- co\_kwonlyargcount (code object attribute), 36
- co\_kwonlyargcount (ιδιότητα της codeobject), 37
- co\_lines() (μέθοδος της codeobject), 38
- co\_lnotab (code object attribute), 36
- co\_lnotab (ιδιότητα της codeobject), 37
- co\_name (code object attribute), 36
- co\_name (ιδιότητα της codeobject), 37
- co\_names (code object attribute), 36
- co\_names (ιδιότητα της codeobject), 37
- co\_nlocals (code object attribute), 36
- co\_nlocals (ιδιότητα της codeobject), 37
- co\_positions() (μέθοδος της codeobject), 38
- co\_posonlyargcount (code object attribute), 36
- co\_posonlyargcount (ιδιότητα της codeobject), 37
- co\_qualname (code object attribute), 36
- co\_qualname (ιδιότητα της codeobject), 37
- co\_stacksize (code object attribute), 36
- co\_stacksize (ιδιότητα της codeobject), 37
- co\_varnames (code object attribute), 36
- co\_varnames (ιδιότητα της codeobject), 37
- code object, 36
- collections
  - module, 26
- comma, 87
  - trailing, 104
- command line, 141
- comment, 8
- comparison, 99
- comparisons, 44
  - chaining, 99
- compile
  - built-in function, 115
- complex
  - built-in function, 59
  - number, 25
  - object, 25
- complex literal, 18
- compound
  - statement, 117
- comprehensions, 87
  - dictionary, 88
  - list, 88
  - set, 88
- conditional
  - expression, 103
- constant, 12
- constructor
  - class, 42
- container, 24, 33
- context, **165**
- context manager, 59
- context μεταβλητή, **165**
- contiguous, **165**
- continue
  - statement, **112**, 118, 121
- conversion
  - arithmetic, 85
  - string, 43, 105
- coroutine, 63, 90, **165**
  - function, 30
- coroutine συνάρτηση, **165**
- dangling
  - else, 118
- data, 23
  - type, 24
  - type, immutable, 86

- dbm.gnu
  - module, 27
- dbm.ndbm
  - module, 27
- debugging
  - assertions, 109
- decimal literal, 18
- decorator, 166
- def
  - statement, 131
- default
  - parameter value, 132
- definition
  - class, 109, 133
  - function, 109, 131
- del
  - statement, 42, 109
- deletion
  - attribute, 109
  - target, 109
  - target list, 109
- delimiters, 21
- descriptor, 166
- destructor, 42, 106
- dictionary
  - comprehensions, 88
  - display, 88
  - object, 27, 33, 44, 88, 94, 107
- display
  - dictionary, 88
  - list, 88
  - set, 88
- division, 98
- divmod
  - built-in function, 58
- docstring, 133, 166
- documentation string, 38
- duck-typing, 166
- dunder, 167
- e
  - in numeric literal, 19
- elif
  - keyword, 118
- else
  - conditional expression, 103
  - dangling, 118
  - keyword, 112, 118, 119, 121
- empty
  - list, 88
  - tuple, 26, 87
- encoding declarations (*source file*), 8
- escape sequence, 13
- eval
  - built-in function, 115, 142
- evaluation
  - order, 104
- exc\_info (*in module sys*), 40
- except
  - keyword, 119
- except\_star
  - keyword, 120
- exception, 110
  - AssertionError, 109
  - AttributeError, 94
  - GeneratorExit, 91, 93
  - ImportError, 112
  - NameError, 85
  - StopAsyncIteration, 93
  - StopIteration, 91, 110
  - TypeError, 97
  - ValueError, 99
  - ZeroDivisionError, 98
- chaining, 111
- handler, 40
- raising, 110
- exclusive
  - or, 99
- exec
  - built-in function, 115
- execution
  - frame, 133
  - stack, 40
- expression, 85
  - Conditional, 102
  - assignment, 102
  - conditional, 103
  - generator, 89
  - lambda, 103, 132
  - list, 103, 105
  - statement, 105
  - yield, 89
- extension
  - module, 24
- f'
  - formatted string literal, 16
- f"
  - formatted string literal, 16
- f-string, 167
- f-strings, 167
- f\_back (*frame attribute*), 39
- f\_back (*ιδιότητα της frame*), 39
- f\_builtins (*frame attribute*), 39
- f\_builtins (*ιδιότητα της frame*), 39
- f\_code (*frame attribute*), 39
- f\_code (*ιδιότητα της frame*), 39
- f\_globals (*frame attribute*), 39
- f\_globals (*ιδιότητα της frame*), 39
- f\_lasti (*frame attribute*), 39
- f\_lasti (*ιδιότητα της frame*), 39
- f\_lineno (*frame attribute*), 39
- f\_lineno (*ιδιότητα της frame*), 40
- f\_locals (*frame attribute*), 39
- f\_locals (*ιδιότητα της frame*), 39
- f\_trace (*frame attribute*), 39
- f\_trace (*ιδιότητα της frame*), 40
- f\_trace\_lines (*frame attribute*), 39



`f_trace_lines` (ιδιότητα της *frame*), 40  
`f_trace_opcodes` (*frame attribute*), 39  
`f_trace_opcodes` (ιδιότητα της *frame*), 40  
`finalizer`, 42  
`finally`  
     keyword, 110, 112, 119, 121  
`find_spec`  
     finder, 76  
`finder`, 75, 168  
     `find_spec`, 76  
`float`  
     built-in function, 59  
`floating-point`  
     number, 25  
     object, 25  
`floating-point literal`, 18  
`for`  
     in comprehensions, 87  
     statement, 112, 118  
`form`  
     `lambda`, 103  
`format()` (*built-in function*)  
     `__str__()` (*object method*), 43  
`formatted string literal`, 16  
`frame`  
     execution, 133  
     object, 39  
`from`  
     import statement, 113  
     keyword, 89, 112  
     yield from expression, 90  
`frozenset`  
     object, 27  
`fstring`, 16  
`f-string`, 16  
`function`  
     annotations, 132  
     anonymous, 103  
     argument, 27  
     call, 27, 96  
     call, user-defined, 96  
     definition, 109, 131  
     generator, 89, 110  
     name, 131  
     object, 27, 30, 96, 131  
     user-defined, 27  
`future`  
     statement, 114  
`garbage collection`, 23  
`generator`, 169  
     expression, 89  
     function, 29, 89, 110  
     iterator, 29, 110  
     object, 38, 89, 90  
`generator iterator`, 169  
`generator έκφραση`, 169  
`generic`  
     special attribute, 24  
`global`  
     name binding, 115  
     namespace, 28  
     statement, 109, 115  
`global interpreter lock`, 169  
`grouping`, 9  
`guard`, 125  
`handler`  
     exception, 40  
`hash`  
     built-in function, 44  
`hash character`, 8  
`hash-based pyc`, 170  
`hashable`, 89, 170  
`hexadecimal literal`, 18  
`hierarchy`  
     type, 24  
`hooks`  
     import, 75  
     meta, 75  
     path, 75  
`id`  
     built-in function, 23  
`identifier`, 10, 85  
`identity`  
     test, 102  
`identity of an object`, 23  
`if`  
     conditional expression, 103  
     in comprehensions, 87  
     keyword, 123  
     statement, 118  
`imaginary literal`, 18  
`immutable`, 170  
     data type, 86  
     object, 26, 86, 89  
`immutable object`, 23  
`immutable sequence`  
     object, 26  
`immutable types`  
     subclassing, 42  
`import`  
     hooks, 75  
     statement, 30, 112  
`import hooks`, 75  
`import machinery`, 73  
`in`  
     keyword, 118  
     operator, 102  
`inclusive`  
     or, 99  
`indentation`, 9  
`index operation`, 26  
`indices()` (*μέθοδος της slice*), 41  
`inheritance`, 133  
`input`, 142  
`instance`  
     call, 55, 97

- class, 35
  - object, 33, 35, 96
- int
  - built-in function, 59
- integer, 26
  - object, 25
  - representation, 25
- integer literal, 18
- interactive mode, 141
- internal type, 35
- interpolated string literal, 16
- interpreted, **171**
- interpreter, 141
- inversion, 97
- invocation, 27
- io
  - module, 35
- irrefutable case block, 125
- is
  - operator, 102
- is not
  - operator, 102
- item
  - sequence, 94
  - string, 94
- item selection, 26
- iterable, **171**
  - unpacking, 103
- iterator, **171**
- j
  - in numeric literal, 20
- key, 88
- key/value pair, 88
- keyword, 11
  - as, 112, 119, 122, 123
  - async, 134
  - await, 97, 134
  - case, **123**
  - elif, 118
  - else, 112, 118, 119, 121
  - except, 119
  - except\_star, 120
  - finally, 110, 112, 119, 121
  - from, 89, 112
  - if, 123
  - in, 118
  - yield, 89
- lambda, **172**
  - expression, 103, 132
  - form, 103
- language
  - C, 24, 25, 30, 99
  - Java, 25
- last\_traceback (*in module sys*), 40
- leading whitespace, 9
- len
  - built-in function, 26, 27, 56
- lexical analysis, 7
- line continuation, 8
- line joining, 7, 8
- line structure, 7
- list
  - assignment, target, 106
  - comprehensions, 88
  - deletion target, 109
  - display, 88
  - empty, 88
  - expression, 103, 105
  - object, 26, 88, 94, 107
  - target, 106, 118
- list comprehension, **172**
- literal, 12, 86
- loader, 75, **172**
- logical line, 7
- loop
  - statement, 112, 118
- loop control
  - target, 112
- magic
  - μέθοδος, 173
- makefile() (*socket method*), 35
- mangling
  - name, 85
- mapping, **173**
  - object, 27, 35, 94, 107
- match
  - case, 123
  - statement, **123**
- matrix multiplication, 98
- membership
  - test, 102
- meta
  - hooks, 75
- meta hooks, 75
- meta path finder, **173**
- metaclass, 50
- metaclass hint, 51
- method
  - built-in, 30
  - call, 96
  - object, 29, 30, 96
  - user-defined, 29
- minus, 97
- module, **173**
  - \_\_main\_\_, 69, 141
  - array, 26
  - builtins, 141
  - collections, 26
  - dbm.gnu, 27
  - dbm.ndbm, 27
  - extension, 24
  - importing, 112
  - io, 35
  - namespace, 30
  - object, 30, 94
  - sys, 120, 141

- module spec, 75
- module επέκτασης, **167**
- modulo, 98
- mro() (μέθοδος της type), 35
- multiplication, 98
- mutable, **173**
  - object, 26, 106, 107
- mutable object, 23
- mutable sequence
  - object, 26
- name, 10, 85
  - binding, 106, 112, 113, 131, 133
  - binding, global, 115
  - class, 133
  - function, 131
  - mangling, 85
  - rebinding, 106
  - unbinding, 109
- named expression, 102
- named tuple, **173**
- names
  - private, 85
- namespace, **174**
  - global, 28
  - module, 30
  - package, 74
- negation, 97
- nested scope, **174**
- nonlocal
  - statement, 115
- not
  - operator, 102
- not in
  - operator, 102
- null
  - operation, 109
- number, 18
  - complex, 25
  - floating-point, 25
- numeric
  - object, 25, 35
- numeric literal, 18
- object, 23
  - Boolean, 25
  - Ellipsis, 24
  - None, 24, 105
  - NotImplemented, 24
  - asynchronous-generator, 93
  - built-in function, 30, 96
  - built-in method, 30, 96
  - callable, 27, 95
  - class, 33, 96, 133
  - class instance, 33, 35, 96
  - code, 36
  - complex, 25
  - dictionary, 27, 33, 44, 88, 94, 107
  - floating-point, 25
  - frame, 39
  - frozenset, 27
  - function, 27, 30, 96, 131
  - generator, 38, 89, 90
  - immutable, 26, 86, 89
  - immutable sequence, 26
  - instance, 33, 35, 96
  - integer, 25
  - list, 26, 88, 94, 107
  - mapping, 27, 35, 94, 107
  - method, 29, 30, 96
  - module, 30, 94
  - mutable, 26, 106, 107
  - mutable sequence, 26
  - numeric, 25, 35
  - sequence, 26, 35, 94, 102, 107, 118
  - set, 27, 88
  - set type, 27
  - slice, 56
  - string, 94
  - traceback, 40, 110, 120
  - tuple, 26, 94, 103
  - user-defined function, 27, 96, 131
  - user-defined method, 29
- object.\_\_match\_args\_\_ (ενσωματωμένη μεταβλητή), 60
- object.\_\_slots\_\_ (ενσωματωμένη μεταβλητή), 49
- octal literal, 18
- open
  - built-in function, 35
- operation
  - Boolean, 102
  - binary arithmetic, 97
  - binary bitwise, 99
  - null, 109
  - power, 97
  - shifting, 99
  - unary arithmetic, 97
  - unary bitwise, 97
- operator
  - (minus), 97, 98
  - % (percent), 98
  - & (ampersand), 99
  - \* (asterisk), 98
  - \*\*, 97
  - + (plus), 97, 98
  - / (slash), 98
  - //, 98
  - < (less), 99
  - <<, 99
  - <=, 99
  - !=, 99
  - ==, 99
  - > (greater), 99
  - >=, 99
  - >>, 99
  - @ (at), 98
  - ^ (caret), 99

- and, 102
- in, 102
- is, 102
- is not, 102
- not, 102
- not in, 102
- or, 102
- overloading, 41
- precedence, 104
- ternary, 103
  - | (*vertical bar*), 99
  - ~ (*tilde*), 97
- operators, 21
- or
  - bitwise, 99
  - exclusive, 99
  - inclusive, 99
  - operator, 102
- ord
  - built-in function, 26
- order
  - evaluation, 104
- output, 105
  - standard, 105
- overloading
  - operator, 41
- package, 74
  - namespace, 74
  - portion, 74
  - regular, 74
- parameter
  - call semantics, 95
  - function definition, 131
  - value, default, 132
- parenthesized form, 87
- parser, 7
- pass
  - statement, 109
- path
  - hooks, 75
- path based finder, 80, 176
- path entry, 175
- path entry finder, 175
- path entry hook, 176
- path hooks, 75
- path-like αντικείμενο, 176
- pattern matching, 123
- physical line, 7, 8, 13
- plus, 97
- popen() (*in module os*), 35
- portion
  - package, 74
- pow
  - built-in function, 58
- power
  - operation, 97
- precedence
  - operator, 104

- primary, 93
- print
  - built-in function, 43
- print() (*built-in function*)
  - \_\_str\_\_() (*object method*), 43
- private
  - names, 85
- procedure
  - call, 105
- program, 141
- provisional API, 176
- provisional πακέτο, 176
- r'
  - raw string literal, 16
- r"
  - raw string literal, 16
- raise
  - statement, 110
- raising
  - exception, 110
- range
  - built-in function, 119
- rebinding
  - name, 106
- reference
  - attribute, 94
- reference counting, 23
- regular
  - package, 74
- relative
  - import, 113
- replace() (*μέθοδος της codeobject*), 39
- repr
  - built-in function, 105
- repr() (*built-in function*)
  - \_\_repr\_\_() (*object method*), 43
- representation
  - integer, 25
- reserved word, 11
- return
  - statement, 109, 121
- round
  - built-in function, 59
- send() (*μέθοδος της coroutine*), 63
- send() (*μέθοδος της generator*), 91
- sequence
  - item, 94
  - object, 26, 35, 94, 102, 107, 118
- set
  - comprehensions, 88
  - display, 88
  - object, 27, 88
- set comprehension, 178
- set type
  - object, 27
- shifting
  - operation, 99
- simple

- statement, 105
- singleton
  - tuple, 26
- slice, 94, **178**
  - built-in function, 41
  - object, 56
- slicing, 26, 94
  - assignment, 107
- soft keyword, 11
- source character set, 8
- space, 9
- special
  - attribute, 24
  - attribute, generic, 24
  - μέθοδος, **178**
- stack
  - execution, 40
  - trace, 40
- standard
  - output, 105
- standard input, 141
- standard library, **178**
- start (*slice object attribute*), 41, 95
- statement
  - assert, **109**
  - assignment, 26, 106
  - assignment, annotated, 108
  - assignment, augmented, 107
  - async def, 134
  - async for, 135
  - async with, 135
  - break, **112**, 118, 121
  - class, 133
  - compound, 117
  - continue, **112**, 118, 121
  - def, 131
  - del, 42, **109**
  - expression, 105
  - for, 112, **118**
  - future, 114
  - global, 109, **115**
  - if, **118**
  - import, 30, **112**
  - loop, 112, 118
  - match, **123**
  - nonlocal, 115
  - pass, 109
  - raise, **110**
  - return, **109**, 121
  - simple, 105
  - try, 40, **119**
  - type, 115
  - while, 112, **118**
  - with, 59, **122**
  - yield, 110
- statement grouping, 9
- stderr (*in module sys*), 35
- stdin (*in module sys*), 35
- stdio, 35
- stdlib, **178**
- stdout (*in module sys*), 35
- step (*slice object attribute*), 41, 95
- stop (*slice object attribute*), 41, 95
- string
  - \_\_format\_\_() (*object method*), 43
  - \_\_str\_\_() (*object method*), 43
  - conversion, 43, 105
  - formatted literal, 16
  - immutable sequences, 26
  - interpolated literal, 16
  - item, 94
  - object, 94
- string literal, 12
- strong reference, **178**
- subclassing
  - immutable types, 42
- subscription, 26, 27, 94
  - assignment, 107
- subtraction, 98
- suite, 117
- sys
  - module, 120, 141
- sys.exc\_info, 40
- sys.exception, 40
- sys.last\_traceback, 40
- sys.meta\_path, 76
- sys.modules, 75
- sys.path, 80
- sys.path\_hooks, 80
- sys.path\_importer\_cache, 80
- sys.stderr, 35
- sys.stdin, 35
- sys.stdout, 35
- t-string, **179**
- t-strings, **179**
- tab, 9
- target, 106
  - deletion, 109
  - list, 106, 118
  - list assignment, 106
  - list, deletion, 109
  - loop control, 112
- tb\_frame (*traceback attribute*), 40
- tb\_frame (*ιδιότητα της traceback*), 41
- tb\_lasti (*traceback attribute*), 40
- tb\_lasti (*ιδιότητα της traceback*), 41
- tb\_lineno (*traceback attribute*), 40
- tb\_lineno (*ιδιότητα της traceback*), 41
- tb\_next (*traceback attribute*), 41
- tb\_next (*ιδιότητα της traceback*), 41
- ternary
  - operator, 103
- test
  - identity, 102
  - membership, 102
- throw() (*μέθοδος της coroutine*), 63

`throw()` (μέθοδος της *generator*), 91  
`token`, 7  
`trace`  
    `stack`, 40  
`traceback`  
    `object`, 40, 110, 120  
`trailing`  
    `comma`, 104  
`triple-quoted string`, 12  
`try`  
    `statement`, 40, 119  
`tuple`  
    `empty`, 26, 87  
    `object`, 26, 94, 103  
    `singleton`, 26  
`type`, 24  
    `built-in function`, 23, 50  
    `data`, 24  
    `hierarchy`, 24  
    `immutable data`, 86  
    `statement`, 115  
`type alias`, 180  
`type hint`, 180  
`type of an object`, 23  
`type parameters`, 136  
`types`, `internal`, 35  
`u'`  
    `string literal`, 13  
`u"`  
    `string literal`, 13  
`unary`  
    `arithmetic operation`, 97  
    `bitwise operation`, 97  
`unbinding`  
    `name`, 109  
`unpacking`  
    `dictionary`, 89  
    `in function calls`, 96  
    `iterable`, 103  
`unreachable object`, 23  
`unrecognized escape sequence`, 15  
`user-defined`  
    `function`, 27  
    `function call`, 96  
    `method`, 29  
`user-defined function`  
    `object`, 27, 96, 131  
`user-defined method`  
    `object`, 29  
`value`, 88  
    `default parameter`, 132  
`value of an object`, 23  
`values`  
    `writing`, 105  
`virtual environment`, 180  
`virtual machine`, 181  
`walrus operator`, 102, 181  
`while`

`statement`, 112, 118  
`with`  
    `statement`, 59, 122  
`writing`  
    `values`, 105  
`xor`  
    `bitwise`, 99  
`yield`  
    `examples`, 91  
    `expression`, 89  
    `keyword`, 89  
    `statement`, 110  
`{ }` (*curly brackets*)  
    `dictionary expression`, 88  
    `in formatted string literal`, 16  
    `set expression`, 88  
`|` (*vertical bar*)  
    `operator`, 99  
`|=`  
    `augmented assignment`, 107  
`~` (*tilde*)  
    `operator`, 97

## A

Αθάνατο, 170  
ακέραια διαίρεση, 168  
ακολουθία, 177  
αναγνωρισμένο όνομα, 177  
αντικείμενο, 174  
αντικείμενο αρχείου, 167  
αντικείμενο που μοιάζει με αρχείο, 167  
αξιολόγηση συνάρτησης, 167  
απαρχαιωμένη με ήπιο τρόπο, 178  
από  
    δήλωση εισαγωγής, 67  
αρχείο κειμένου, 179  
ασύγχρονος `generator`, 162  
ασύγχρονος `generator iterator`, 162  
ασύγχρονος `iterable`, 162  
ασύγχρονος `iterator`, 162  
ασύγχρονος διαχειριστής `context`, 162  
αφηρημένη βασική κλάση, 161

## B

βελτιστοποιημένο πεδίο ορατότητας  
    (*scope*), 174

## Γ

γενική συνάρτηση, 169  
γενικός τύπος, 169  
γραμματική, 4

## Δ

δανεική αναφορά, 163  
δήλωση, 178  
διαδραστικός, 170  
διαχειρίζεται μια εξαίρεση, 71  
διαχείριση σφαλμάτων, 71

διαχειριστής context, **165**  
 διαχειριστής εξαιρέσεων, **71**  
 δυαδικό αρχείο, **163**  
 δωρεάν μεταβλητή, **168**  
 δωρεάν νήμα, **168**

## Ε

ειδική μέθοδος, **178**  
 εισαγόμενο path, **170**  
 εισαγωγή, **170**  
 εισαγωγή, **170**  
 εκτέλεση  
   περιορισμένη, **71**  
   πλαίσιο, **67**  
 έκφραση, **167**  
 ελεγκτής στατικού τύπου, **178**  
 ελεύθερη  
   μεταβλητή, **68**  
 εξαίρεση, **71**

## Κ

καθολικές νέες γραμμές, **180**  
 κάνει raise μια εξαίρεση, **71**  
 κανονικό πακέτο, **177**  
 κατανόηση λεξικού, **166**  
 κατάσταση νήματος, **179**  
 κατάσταση συνδεδεμένου νήματος, **163**  
 κλάση, **164**  
 κλάση νέου στυλ, **174**  
 κυκλική απομόνωση, **165**  
 κώδικας  
   μπλοκ, **67**  
 κωδικοποίηση κειμένου, **179**  
 κωδικοποίηση συστήματος αρχείων και  
   χειριστής σφαλμάτων, **167**

## Λ

λεκτικό σύμβολο (*token*), **179**  
 λεξικό, **166**  
 λεξικός αναλυτής, **172**  
 λεξιλογικοί ορισμοί, **5**  
 λίστα, **172**

## Μ

μαγική μέθοδος, **173**  
 μέθοδος, **173**  
   magic, **173**  
   special, **178**  
 μετα-κλάση, **173**  
 μεταβλητή  
   ελεύθερη, **68**  
 μεταβλητή κλάσης, **164**  
 μεταβλητή κλεισίματος, **164**  
 μεταβλητή περιβάλλοντος  
   PYTHONHASHSEED, **45**  
   PYTHONNODEBUGRANGES, **38**  
   PYTHONPATH, **80**  
   PYTHON\_GIL, **169**

μιγαδικός αριθμός, **164**  
 μοναδικό dispatch, **178**  
 μοντέλο εκτέλεσης, **67**  
 μοντέλο τερματισμού, **71**  
 μπλοκ, **67**  
   κώδικας, **67**

## Ο

όνομα, **67**  
   σύνδεση, **67**  
 όρισμα, **162**  
 όρισμα keyword, **172**  
 όρισμα θέσης, **176**  
 όψη λεξικού, **166**

## Π

πακέτο, **175**  
 πακέτο namespace, **174**  
 παράμετρος, **175**  
 πεδίο, **67**, **68**  
 περιβάλλον, **68**  
 περιορισμένη  
   εκτέλεση, **71**  
 πλαίσιο  
   εκτέλεση, **67**  
 πλήθος αναφοράς, **177**  
 πρωτόκολλο διαχείρισης περιβάλλοντος,  
   **165**

## Σ

σειρά ανάλυσης μεθόδων, **173**  
 σημειογραφία, **4**  
 συλλογή απορριμάτων, **168**  
 συμβολοσειρά τριπλών εισαγωγικών, **179**  
 συνάρτηση, **168**  
 συνάρτηση annotate, **161**  
 συνάρτηση annotation, **168**  
 συνάρτηση key, **171**  
 σύνδεση  
   όνομα, **67**  
 συντακτικό, **4**  
 σφάλματα, **71**

## Τ

τερματισμός λειτουργίας διερμηνέα, **171**  
 τεχνικές προδιαγραφές module, **173**  
 τμήμα, **176**  
 τοπική κωδικοποίηση, **172**  
 τρέχον πλαίσιο, **165**  
 τύπος, **180**

## Χ

χαρακτηριστικό, **163**  
 χώρος ονομάτων, **67**