

---

# Python Frequently Asked Questions

*Δημοσίευση 3.11.14*

Guido van Rossum and the Python development team

Οκτωβρίου 15, 2025

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



<b>1</b>	<b>General Python FAQ</b>	<b>1</b>
1.1	General Information	1
1.1.1	What is Python?	1
1.1.2	What is the Python Software Foundation?	1
1.1.3	Are there copyright restrictions on the use of Python?	2
1.1.4	Why was Python created in the first place?	2
1.1.5	What is Python good for?	2
1.1.6	How does the Python version numbering scheme work?	3
1.1.7	How do I obtain a copy of the Python source?	3
1.1.8	How do I get documentation on Python?	3
1.1.9	I've never programmed before. Is there a Python tutorial?	4
1.1.10	Is there a newsgroup or mailing list devoted to Python?	4
1.1.11	How do I get a beta test version of Python?	4
1.1.12	How do I submit bug reports and patches for Python?	4
1.1.13	Are there any published articles about Python that I can reference?	4
1.1.14	Are there any books on Python?	4
1.1.15	Where in the world is <a href="http://www.python.org">www.python.org</a> located?	5
1.1.16	Why is it called Python?	5
1.1.17	Do I have to like «Monty Python's Flying Circus»?	5
1.2	Python in the real world	5
1.2.1	How stable is Python?	5
1.2.2	How many people are using Python?	5
1.2.3	Have any significant projects been done in Python?	5
1.2.4	What new developments are expected for Python in the future?	6
1.2.5	Is it reasonable to propose incompatible changes to Python?	6
1.2.6	Is Python a good language for beginning programmers?	6
<b>2</b>	<b>FAQ Προγραμματισμού</b>	<b>9</b>
2.1	Γενικές Ερωτήσεις	9
2.1.1	Υπάρχει πρόγραμμα εντοπισμού σφαλμάτων σε επίπεδο πηγαίου κώδικα με σημεία διακοπής, με ένα βήμα κλπ.;	9
2.1.2	Υπάρχουν εργαλεία που βοηθούν στην εύρεση σφαλμάτων ή στην εκτέλεση στατικής ανάλυσης;	10
2.1.3	Πως μπορώ να δημιουργήσω ένα stand-alone binary από ένα Python script;	10
2.1.4	Υπάρχουν πρότυπα κωδικοποίησης ή οδηγός στυλ για προγράμματα Python;	11
2.2	Βασική Γλώσσα	11

2.2.1	Γιατί λαμβάνω ένα UnboundLocalError όταν η μεταβλητή έχει μια τιμή; . . . . .	11
2.2.2	Ποιοι είναι οι κανόνες για τις τοπικές και τις καθολικές μεταβλητές στην Python; . . . .	12
2.2.3	Γιατί τα lambdas που ορίζονται σε έναν βρόχο με διαφορετικές τιμές επιστρέφουν όλα το ίδιο αποτέλεσμα; . . . . .	12
2.2.4	Πώς μοιράζομαι καθολικές μεταβλητές σε modules; . . . . .	13
2.2.5	Ποιες είναι οι «βέλτιστες πρακτικές» για τη χρήση import σε ένα module; . . . . .	14
2.2.6	Γιατί μοιράζονται οι προεπιλεγμένες τιμές μεταξύ των αντικειμένων; . . . . .	14
2.2.7	Πώς μπορώ να μεταβιβάσω προαιρετικές παραμέτρους ή παραμέτρους λέξεων-κλειδιών από τη μια συνάρτηση στην άλλη; . . . . .	15
2.2.8	Ποια είναι η διαφορά μεταξύ ορισμάτων και παραμέτρων; . . . . .	16
2.2.9	Γιατί η αλλαγή της λίστας “y” αλλάζει επίσης και τη λίστα “x”; . . . . .	16
2.2.10	Πώς μπορώ να γράψω μια συνάρτηση με παραμέτρους εξόδου (κλήση με αναφορά); . .	17
2.2.11	Πώς δημιουργείτε μια συνάρτηση υψηλότερης τάξης στην Python; . . . . .	18
2.2.12	Πώς μπορώ να αντιγράψω ένα αντικείμενο στην Python; . . . . .	19
2.2.13	Πώς μπορώ να βρω τις μεθόδους ή τα χαρακτηριστικά ενός αντικειμένου; . . . . .	20
2.2.14	Πώς μπορεί ο κώδικας μου να ανακαλύψει το όνομα ενός αντικειμένου; . . . . .	20
2.2.15	Τι συμβαίνει με την προτεραιότητα του τελεστή κόμματος; . . . . .	20
2.2.16	Υπάρχει ισοδύναμο του τριαδικού τελεστή «?:» της C; . . . . .	21
2.2.17	Είναι δυνατόν να γράψουμε ασαφή one-liners στην Python; . . . . .	21
2.2.18	Τι σημαίνει η κάθετος(/) στη λίστα παραμέτρων μιας συνάρτησης; . . . . .	22
2.3	Αριθμοί και συμβολοσειρές . . . . .	22
2.3.1	Πώς προσδιορίζω δεκαεξαδικούς ή οκταδικούς ακέραιους αριθμούς; . . . . .	22
2.3.2	Γιατί το -22 // 10 επιστρέφει -3; . . . . .	23
2.3.3	Πώς μπορώ να πάρω το literal χαρακτηριστικό int αντί για το SyntaxError; . . . . .	23
2.3.4	Πώς μετατρέπω μια συμβολοσειρά σε έναν αριθμό; . . . . .	23
2.3.5	Πώς μετατρέπω έναν αριθμό σε συμβολοσειρά; . . . . .	24
2.3.6	Πώς μπορώ να τροποποιήσω μια συμβολοσειρά στη θέση της; . . . . .	24
2.3.7	Πώς μπορώ να χρησιμοποιήσω συμβολοσειρές για να καλέσω συναρτήσεις/μεθόδους; .	24
2.3.8	Is there an equivalent to Perl’s chomp() for removing trailing newlines from strings? . . . . .	25
2.3.9	Is there a scanf() or sscanf() equivalent? . . . . .	26
2.3.10	What does “UnicodeDecodeError” or “UnicodeEncodeError” error mean? . . . . .	26
2.3.11	Μπορώ να τερματίσω μια ακατέργαστη συμβολοσειρά με περιττό αριθμό backslashes; . .	26
2.4	Απόδοση . . . . .	27
2.4.1	Το πρόγραμμα μου είναι πολύ αργό. Πώς μπορώ να το επιταχύνω; . . . . .	27
2.4.2	Ποιος είναι ο πιο αποτελεσματικός τρόπος για να συνδέσετε πολλές συμβολοσειρές με- ταξύ τους; . . . . .	28
2.5	Ακολουθίες (Πλειάδες/Λίστες) . . . . .	28
2.5.1	Πώς μπορώ να κάνω μετατροπή μεταξύ πλειάδων και λιστών; . . . . .	28
2.5.2	Τι είναι αρνητικός δείκτης; . . . . .	28
2.5.3	Πώς μπορώ να επαναλάβω μια ακολουθία με αντίστροφη σειρά; . . . . .	29
2.5.4	Πώς αφαιρείτε διπλότυπα από μια λίστα; . . . . .	29
2.5.5	Πώς αφαιρείτε πολλαπλά στοιχεία από μία λίστα . . . . .	29
2.5.6	Πώς μπορείτε να φτιάξετε έναν πίνακα στην Python; . . . . .	30
2.5.7	Πώς φτιάχνω μια πολυδιάστατη λίστα; . . . . .	30
2.5.8	Πώς μπορώ να εφαρμόσω μια μέθοδο ή μια συνάρτηση σε μια ακολουθία αντικειμένων; .	31
2.5.9	Γιατί το a_tuple[i] += [“item”] δημιουργεί μια εξαίρεση όταν λειτουργεί η προσθήκη; . .	31
2.5.10	Θέλω να κάνω μια περίπλοκη ταξινόμηση: μπορείτε να κάνετε ένα Schwartzian Transform στην Python; . . . . .	32
2.5.11	Πώς μπορώ να ταξινομήσω μια λίστα με βάση τις τιμές από μια άλλη λίστα; . . . . .	33
2.6	Αντικείμενα . . . . .	33
2.6.1	Τι είναι μια κλάση; . . . . .	33
2.6.2	Τι είναι μια μέθοδος; . . . . .	33
2.6.3	Τι είναι το self; . . . . .	33

2.6.4	Πώς μπορώ να ελέγξω εάν ένα αντικείμενο είναι μια οντότητα μιας δεδομένης κλάσης ή μιας υποκλάσης της; . . . . .	34
2.6.5	Τι είναι το <code>delegation</code> ; . . . . .	35
2.6.6	Πώς μπορώ να καλέσω μια μέθοδο που ορίζεται σε μια βασική κλάση από μια παράγωγη κλάση που την επεκτείνει; . . . . .	36
2.6.7	Πως μπορώ να οργανώσω τον κώδικα μου προκειμένου να διευκολύνω την αλλαγή της βασικής κλάσης; . . . . .	36
2.6.8	Πως δημιουργώ δεδομένα στατικής κλάσης και μεθόδους στατικής κλάσης; . . . . .	36
2.6.9	Πως μπορώ να υπερφορτώσω κατασκευαστές (ή μεθόδους) στην Python; . . . . .	37
2.6.10	Προσπαθώ να χρησιμοποιήσω <code>__spam</code> και λαμβάνω ένα σφάλμα σχετικά με το <code>__SomeClassName__spam</code> . . . . .	38
2.6.11	Η κλάση μου ορίζει <code>__del__</code> αλλά δεν καλείται όταν διαγράφω το αντικείμενο. . . . .	38
2.6.12	Πως μπορώ να λάβω μια λίστα με όλες τις οντότητες μιας δεδομένης κλάσης; . . . . .	38
2.6.13	Γιατί το αποτέλεσμα του <code>id()</code> φαίνεται να μην είναι μοναδικό; . . . . .	39
2.6.14	Πότε μπορώ να βασιστώ σε δοκιμές ταυτότητας με τον τελεστή <code>is</code> ; . . . . .	39
2.6.15	Πώς μπορεί μια υποκλάση να ελέγξει ποια δεδομένα αποθηκεύονται σε μια αμετάβλητη παρουσία; . . . . .	40
2.6.16	Πώς μπορώ να αποθηκεύσω τις κλήσεις μεθόδου στην κρυφή μνήμη; . . . . .	41
2.7	<b>Modules</b> . . . . .	42
2.7.1	Πως δημιουργώ ένα <code>.pyc</code> αρχείο; . . . . .	42
2.7.2	Πως μπορώ να βρω το όνομα του τρέχοντος module; . . . . .	43
2.7.3	Πως μπορώ να έχω modules που εισάγουν αμοιβαία το ένα το άλλο; . . . . .	44
2.7.4	<code>__import__</code> ("x.y.z") επιστρέφει <code>&lt;module "x"&gt;</code> • πως μπορώ να πάρω το <code>z</code> ? . . . . .	45
2.7.5	Όταν επεξεργάζομαι ένα module που έχει εισαχθεί και την επανεισάγω, οι αλλαγές δεν εμφανίζονται. Γιατί συμβαίνει αυτό; . . . . .	45
<b>3</b>	<b>Design and History FAQ</b> . . . . .	<b>47</b>
3.1	Why does Python use indentation for grouping of statements? . . . . .	47
3.2	Why am I getting strange results with simple arithmetic operations? . . . . .	48
3.3	Why are floating-point calculations so inaccurate? . . . . .	48
3.4	Why are Python strings immutable? . . . . .	48
3.5	Why must "self" be used explicitly in method definitions and calls? . . . . .	49
3.6	Why can't I use an assignment in an expression? . . . . .	49
3.7	Why does Python use methods for some functionality (e.g. <code>list.index()</code> ) but functions for other (e.g. <code>len(list)</code> )? . . . . .	49
3.8	Why is <code>join()</code> a string method instead of a list or tuple method? . . . . .	50
3.9	How fast are exceptions? . . . . .	50
3.10	Why isn't there a switch or case statement in Python? . . . . .	51
3.11	Can't you emulate threads in the interpreter instead of relying on an OS-specific thread implementation? . . . . .	51
3.12	Why can't lambda expressions contain statements? . . . . .	51
3.13	Can Python be compiled to machine code, C or some other language? . . . . .	52
3.14	How does Python manage memory? . . . . .	52
3.15	Why doesn't CPython use a more traditional garbage collection scheme? . . . . .	52
3.16	Why isn't all memory freed when CPython exits? . . . . .	53
3.17	Why are there separate tuple and list data types? . . . . .	53
3.18	How are lists implemented in CPython? . . . . .	53
3.19	How are dictionaries implemented in CPython? . . . . .	53
3.20	Why must dictionary keys be immutable? . . . . .	54
3.21	Why doesn't <code>list.sort()</code> return the sorted list? . . . . .	55
3.22	How do you specify and enforce an interface spec in Python? . . . . .	55
3.23	Why is there no <code>goto</code> ? . . . . .	56
3.24	Why can't raw strings (r-strings) end with a backslash? . . . . .	56
3.25	Why doesn't Python have a «with» statement for attribute assignments? . . . . .	57
3.26	Why don't generators support the with statement? . . . . .	58

3.27	Why are colons required for the if/while/def/class statements?	58
3.28	Why does Python allow commas at the end of lists and tuples?	58
<b>4</b>	<b>Library and Extension FAQ</b>	<b>61</b>
4.1	General Library Questions	61
4.1.1	How do I find a module or application to perform task X?	61
4.1.2	Where is the math.py (socket.py, regex.py, etc.) source file?	61
4.1.3	How do I make a Python script executable on Unix?	62
4.1.4	Is there a curses/termcap package for Python?	62
4.1.5	Is there an equivalent to C's onexit() in Python?	62
4.1.6	Why don't my signal handlers work?	63
4.2	Common tasks	63
4.2.1	How do I test a Python program or component?	63
4.2.2	How do I create documentation from doc strings?	64
4.2.3	How do I get a single keypress at a time?	64
4.3	Threads	64
4.3.1	How do I program using threads?	64
4.3.2	None of my threads seem to run: why?	64
4.3.3	How do I parcel out work among a bunch of worker threads?	65
4.3.4	What kinds of global value mutation are thread-safe?	66
4.3.5	Can't we get rid of the Global Interpreter Lock?	67
4.4	Input and Output	67
4.4.1	How do I delete a file? (And other file questions...)	67
4.4.2	How do I copy a file?	68
4.4.3	How do I read (or write) binary data?	68
4.4.4	I can't seem to use os.read() on a pipe created with os.popen(); why?	68
4.4.5	How do I access the serial (RS232) port?	68
4.4.6	Why doesn't closing sys.stdout (stdin, stderr) really close it?	69
4.5	Network/Internet Programming	69
4.5.1	What WWW tools are there for Python?	69
4.5.2	How can I mimic CGI form submission (METHOD=POST)?	69
4.5.3	What module should I use to help with generating HTML?	70
4.5.4	How do I send mail from a Python script?	70
4.5.5	How do I avoid blocking in the connect() method of a socket?	71
4.6	Databases	71
4.6.1	Are there any interfaces to database packages in Python?	71
4.6.2	How do you implement persistent objects in Python?	71
4.7	Mathematics and Numerics	71
4.7.1	How do I generate random numbers in Python?	71
<b>5</b>	<b>Συχνές ερωτήσεις επέκτασης/ενσωμάτωσης</b>	<b>73</b>
5.1	Μπορώ να δημιουργήσω τις δικές μου συναρτήσεις στη C;	73
5.2	Μπορώ να δημιουργήσω τις δικές μου συναρτήσεις στη C++;	73
5.3	Το να γράψει C κάποιος είναι δύσκολο· υπάρχουν άλλες εναλλακτικές;	73
5.4	Πως μπορώ να εκτελέσω αυθαίρετες δηλώσεις Python από το C;	74
5.5	Πώς μπορώ να αξιολογήσω μια αυθαίρετη έκφραση Python από τη C;	74
5.6	Πως μπορώ να εξάγω τιμές C από ένα αντικείμενο Python;	74
5.7	Πώς μπορώ να χρησιμοποιήσω την Py_BuildValue() για να δημιουργήσω μια πλειάδα (tuple) αυθαίρετου μήκους;	74
5.8	Πώς καλώ τη μέθοδο ενός αντικειμένου από τη C;	75
5.9	Πώς μπορώ να κάνω catch την έξοδο από την PyErr_Print() (ή οτιδήποτε εκτυπώνεται σε stdout/stderr);	75
5.10	Πως μπορώ να αποκτήσω πρόσβαση σε ένα module γραμμένο σε Python από τη C;	76
5.11	Πως διασυνδέομαι με αντικείμενα C++ από την Python;	76

5.12	Πρόσθεσα ένα module χρησιμοποιώντας το αρχείο Setup και το make αποτυγχάνει· γιατί;	76
5.13	Πώς κάνω debug μια επέκταση;	77
5.14	Θέλω να κάνω compile ένα Python module στο σύστημα Linux μου, αλλά λείπουν ορισμένα αρχεία. Γιατί;	77
5.15	Πώς μπορώ να ξεχωρίσω την «ελλιπή εισαγωγή» από την «έγκυρη εισαγωγή»;	77
5.16	Πώς μπορώ να βρω απροσδιόριστα σύμβολα g++ __builtin_new ή __pure_virtual;	78
5.17	Μπορώ να δημιουργήσω μια κλάση αντικειμένου με ορισμένες μεθόδους που υλοποιούνται στη C και άλλες στη Python (π.χ. μέσω κληρονομικότητας);	78
<b>6</b>	<b>Python στα Windows FAQ</b>	<b>79</b>
6.1	Πώς μπορώ να εκτελέσω ένα πρόγραμμα Python στα Windows;	79
6.2	Πώς κάνω τα Python scripts εκτελέσιμα;	80
6.3	Γιατί μερικές φορές η Python αργεί τόσο πολύ να ξεκινήσει;	81
6.4	Πώς μπορώ να δημιουργήσω ένα εκτελέσιμο από ένα σενάριο Python;	81
6.5	Είναι ένα αρχείο *.pyd ίδιο με ένα αρχείο DLL;	81
6.6	Πώς μπορώ να ενσωματώσω την Python σε μια εφαρμογή Windows;	81
6.7	Πώς μπορώ να εμποδίσω τους editors να εισάγουν tabs στον πηγαίο της Python μου;	83
6.8	Πώς μπορώ να ελέγξω για ένα πάτημα πλήκτρων χωρίς αποκλεισμό;	83
6.9	Πώς μπορώ να διορθώσω το σφάλμα που λείπει το api-ms-win-crt-runtime-11-1-0.dll;	83
<b>7</b>	<b>Graphic User Interface FAQ</b>	<b>85</b>
7.1	General GUI Questions	85
7.2	What GUI toolkits exist for Python?	85
7.3	Tkinter questions	85
7.3.1	How do I freeze Tkinter applications?	85
7.3.2	Can I have Tk events handled while waiting for I/O?	86
7.3.3	I can't get key bindings to work in Tkinter: why?	86
<b>8</b>	<b>«Γιατί είναι εγκατεστημένη η Python στον υπολογιστή μου;» FAQ</b>	<b>87</b>
8.1	Τι είναι η Python;	87
8.2	Γιατί είναι εγκατεστημένη η Python στον υπολογιστή μου;	87
8.3	Μπορώ να διαγράψω την Python;	88
<b>A'</b>	<b>Γλωσσάρι</b>	<b>89</b>
<b>B'</b>	<b>About these documents</b>	<b>107</b>
B'.1	Contributors to the Python Documentation	107
<b>Γ'</b>	<b>Ιστορία και Άδεια</b>	<b>109</b>
Γ'.1	Η ιστορία του λογισμικού	109
Γ'.2	Όροι και προϋποθέσεις για την πρόσβαση ή την χρήση της Python με άλλους τρόπους	110
Γ'.2.1	PSF LICENSE AGREEMENT FOR PYTHON 3.11.14	110
Γ'.2.2	ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ BEOPEN.COM ΓΙΑ PYTHON 2.0	111
Γ'.2.3	ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ CNRI ΓΙΑ PYTHON 1.6.1	112
Γ'.2.4	ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ CWI ΓΙΑ PYTHON 0.9.0 ΕΩΣ 1.2	113
Γ'.2.5	ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.11.14 DOCUMENTATION	114
Γ'.3	Άδειες και Ευχαριστίες για Ενσωματωμένο Λογισμικό	114
Γ'.3.1	Mersenne Twister	114
Γ'.3.2	Sockets	115
Γ'.3.3	Ασύγχρονες socket υπηρεσίες	116
Γ'.3.4	Διαχείριση Cookie	116
Γ'.3.5	Ανίχνευση εκτέλεσης	117
Γ'.3.6	Συναρτήσεις UUencode και UUdecode	117
Γ'.3.7	Κλήσεις Απομακρυσμένης Διαδικασίας XML	118
Γ'.3.8	test_epoll	118

Γ'.3.9	Επιλογή kqueue . . . . .	119
Γ'.3.10	SipHash24 . . . . .	119
Γ'.3.11	strtod και dtoa . . . . .	120
Γ'.3.12	OpenSSL . . . . .	120
Γ'.3.13	expat . . . . .	124
Γ'.3.14	libffi . . . . .	124
Γ'.3.15	zlib . . . . .	125
Γ'.3.16	cfuhash . . . . .	125
Γ'.3.17	libmpdec . . . . .	126
Γ'.3.18	W3C C14N σουίτα δοκιμής . . . . .	126
Γ'.3.19	Audioop . . . . .	127
Γ'.3.20	asyncio . . . . .	128
<b>Δ' Copyright</b>		<b>129</b>
<b>Ευρετήριο</b>		<b>131</b>



## 1.1 General Information

### 1.1.1 What is Python?

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. It supports multiple programming paradigms beyond object-oriented programming, such as procedural and functional programming. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants including Linux and macOS, and on Windows.

To find out more, start with [tutorial-index](#). The [Beginner's Guide to Python](#) links to other introductory tutorials and resources for learning Python.

### 1.1.2 What is the Python Software Foundation?

The Python Software Foundation is an independent non-profit organization that holds the copyright on Python versions 2.1 and newer. The PSF's mission is to advance open source technology related to the Python programming language and to publicize the use of Python. The PSF's home page is at <https://www.python.org/psf/>.

Donations to the PSF are tax-exempt in the US. If you use Python and find it helpful, please contribute via [the PSF donation page](#).

### 1.1.3 Are there copyright restrictions on the use of Python?

You can do anything you want with the source, as long as you leave the copyrights in and display those copyrights in any documentation about Python that you produce. If you honor the copyright rules, it's OK to use Python for commercial use, to sell copies of Python in source or binary form (modified or unmodified), or to sell products that incorporate Python in some form. We would still like to know about all commercial use of Python, of course.

See [the license page](#) to find further explanations and the full text of the PSF License.

The Python logo is trademarked, and in certain cases permission is required to use it. Consult [the Trademark Usage Policy](#) for more information.

### 1.1.4 Why was Python created in the first place?

Here's a *very* brief summary of what started it all, written by Guido van Rossum:

I had extensive experience with implementing an interpreted language in the ABC group at CWI, and from working with this group I had learned a lot about language design. This is the origin of many Python features, including the use of indentation for statement grouping and the inclusion of very-high-level data types (although the details are all different in Python).

I had a number of gripes about the ABC language, but also liked many of its features. It was impossible to extend the ABC language (or its implementation) to remedy my complaints – in fact its lack of extensibility was one of its biggest problems. I had some experience with using Modula-2+ and talked with the designers of Modula-3 and read the Modula-3 report. Modula-3 is the origin of the syntax and semantics used for exceptions, and some other Python features.

I was working in the Amoeba distributed operating system group at CWI. We needed a better way to do system administration than by writing either C programs or Bourne shell scripts, since Amoeba had its own system call interface which wasn't easily accessible from the Bourne shell. My experience with error handling in Amoeba made me acutely aware of the importance of exceptions as a programming language feature.

It occurred to me that a scripting language with a syntax like ABC but with access to the Amoeba system calls would fill the need. I realized that it would be foolish to write an Amoeba-specific language, so I decided that I needed a language that was generally extensible.

During the 1989 Christmas holidays, I had a lot of time on my hand, so I decided to give it a try. During the next year, while still mostly working on it in my own time, Python was used in the Amoeba project with increasing success, and the feedback from colleagues made me add many early improvements.

In February 1991, after just over a year of development, I decided to post to USENET. The rest is in the `Misc/HISTORY` file.

### 1.1.5 What is Python good for?

Python is a high-level general-purpose programming language that can be applied to many different classes of problems.

The language comes with a large standard library that covers areas such as string processing (regular expressions, Unicode, calculating differences between files), internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP), software engineering (unit testing, logging, profiling, parsing Python code), and operating system interfaces (system calls, filesystems, TCP/IP sockets). Look at the table of contents for [library-index](#) to get an idea of what's available. A wide variety of third-party extensions are also available. Consult [the Python Package Index](#) to find packages of interest to you.

### 1.1.6 How does the Python version numbering scheme work?

Python versions are numbered «A.B.C» or «A.B»:

- *A* is the major version number – it is only incremented for really major changes in the language.
- *B* is the minor version number – it is incremented for less earth-shattering changes.
- *C* is the micro version number – it is incremented for each bugfix release.

Not all releases are bugfix releases. In the run-up to a new feature release, a series of development releases are made, denoted as alpha, beta, or release candidate. Alphas are early releases in which interfaces aren't yet finalized; it's not unexpected to see an interface change between two alpha releases. Betas are more stable, preserving existing interfaces but possibly adding new modules, and release candidates are frozen, making no changes except as needed to fix critical bugs.

Alpha, beta and release candidate versions have an additional suffix:

- The suffix for an alpha version is «a*N*» for some small number *N*.
- The suffix for a beta version is «b*N*» for some small number *N*.
- The suffix for a release candidate version is «rc*N*» for some small number *N*.

In other words, all versions labeled *2.0a*N** precede the versions labeled *2.0b*N**, which precede versions labeled *2.0rc*N**, and *those* precede 2.0.

You may also find version numbers with a «+» suffix, e.g. «2.2+». These are unreleased versions, built directly from the CPython development repository. In practice, after a final minor release is made, the version is incremented to the next minor version, which becomes the «a0» version, e.g. «2.4a0».

See the [Developer's Guide](#) for more information about the development cycle, and [PEP 387](#) to learn more about Python's backward compatibility policy. See also the documentation for `sys.version`, `sys.hexversion`, and `sys.version_info`.

### 1.1.7 How do I obtain a copy of the Python source?

The latest Python source distribution is always available from python.org, at <https://www.python.org/downloads/>. The latest development sources can be obtained at <https://github.com/python/cpython/>.

The source distribution is a gzipped tar file containing the complete C source, Sphinx-formatted documentation, Python library modules, example programs, and several useful pieces of freely distributable software. The source will compile and run out of the box on most UNIX platforms.

Consult the [Getting Started section of the Python Developer's Guide](#) for more information on getting the source code and compiling it.

### 1.1.8 How do I get documentation on Python?

The standard documentation for the current stable version of Python is available at <https://docs.python.org/3/>. PDF, plain text, and downloadable HTML versions are also available at <https://docs.python.org/3/download.html>.

The documentation is written in reStructuredText and processed by the [Sphinx documentation tool](#). The reStructuredText source for the documentation is part of the Python source distribution.

### 1.1.9 I've never programmed before. Is there a Python tutorial?

There are numerous tutorials and books available. The standard documentation includes [tutorial-index](#).

Consult [the Beginner's Guide](#) to find information for beginning Python programmers, including lists of tutorials.

### 1.1.10 Is there a newsgroup or mailing list devoted to Python?

There is a newsgroup, `comp.lang.python`, and a mailing list, `python-list`. The newsgroup and mailing list are gatewayed into each other – if you can read news it's unnecessary to subscribe to the mailing list. `comp.lang.python` is high-traffic, receiving hundreds of postings every day, and Usenet readers are often more able to cope with this volume.

Announcements of new software releases and events can be found in `comp.lang.python.announce`, a low-traffic moderated list that receives about five postings per day. It's available as [the python-announce mailing list](#).

More info about other mailing lists and newsgroups can be found at <https://www.python.org/community/lists/>.

### 1.1.11 How do I get a beta test version of Python?

Alpha and beta releases are available from <https://www.python.org/downloads/>. All releases are announced on the `comp.lang.python` and `comp.lang.python.announce` newsgroups and on the Python home page at <https://www.python.org/>; an RSS feed of news is available.

You can also access the development version of Python through Git. See [The Python Developer's Guide](#) for details.

### 1.1.12 How do I submit bug reports and patches for Python?

To report a bug or submit a patch, use the issue tracker at <https://github.com/python/cpython/issues>.

For more information on how Python is developed, consult [the Python Developer's Guide](#).

### 1.1.13 Are there any published articles about Python that I can reference?

It's probably best to cite your favorite book about Python.

The [very first article](#) about Python was written in 1991 and is now quite outdated.

Guido van Rossum and Jelke de Boer, «Interactively Testing Remote Servers Using the Python Programming Language», *CWI Quarterly*, Volume 4, Issue 4 (December 1991), Amsterdam, pp 283–303.

### 1.1.14 Are there any books on Python?

Yes, there are many, and more are being published. See the python.org wiki at <https://wiki.python.org/moin/PythonBooks> for a list.

You can also search online bookstores for «Python» and filter out the Monty Python references; or perhaps search for «Python» and «language».

### 1.1.15 Where in the world is [www.python.org](http://www.python.org) located?

The Python project's infrastructure is located all over the world and is managed by the Python Infrastructure Team. Details [here](#).

### 1.1.16 Why is it called Python?

When he began implementing Python, Guido van Rossum was also reading the published scripts from «[Monty Python's Flying Circus](#)», a BBC comedy series from the 1970s. Van Rossum thought he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.

### 1.1.17 Do I have to like «Monty Python's Flying Circus»?

No, but it helps. :)

## 1.2 Python in the real world

### 1.2.1 How stable is Python?

Very stable. New, stable releases have been coming out roughly every 6 to 18 months since 1991, and this seems likely to continue. As of version 3.9, Python will have a new feature release every 12 months ([PEP 602](#)).

The developers issue bugfix releases of older versions, so the stability of existing releases gradually improves. Bugfix releases, indicated by a third component of the version number (e.g. 3.5.3, 3.6.2), are managed for stability; only fixes for known problems are included in a bugfix release, and it's guaranteed that interfaces will remain the same throughout a series of bugfix releases.

The latest stable releases can always be found on the [Python download page](#). There are two production-ready versions of Python: 2.x and 3.x. The recommended version is 3.x, which is supported by most widely used libraries. Although 2.x is still widely used, [it is not maintained anymore](#).

### 1.2.2 How many people are using Python?

There are probably millions of users, though it's difficult to obtain an exact count.

Python is available for free download, so there are no sales figures, and it's available from many different sites and packaged with many Linux distributions, so download statistics don't tell the whole story either.

The [comp.lang.python](#) newsgroup is very active, but not all Python users post to the group or even read it.

### 1.2.3 Have any significant projects been done in Python?

See <https://www.python.org/about/success> for a list of projects that use Python. Consulting the proceedings for [past Python conferences](#) will reveal contributions from many different companies and organizations.

High-profile Python projects include [the Mailman mailing list manager](#) and [the Zope application server](#). Several Linux distributions, most notably [Red Hat](#), have written part or all of their installer and system administration software in Python. Companies that use Python internally include Google, Yahoo, and Lucasfilm Ltd.

### 1.2.4 What new developments are expected for Python in the future?

See <https://peps.python.org/> for the Python Enhancement Proposals (PEPs). PEPs are design documents describing a suggested new feature for Python, providing a concise technical specification and a rationale. Look for a PEP titled «Python X.Y Release Schedule», where X.Y is a version that hasn't been publicly released yet.

New development is discussed on [the python-dev mailing list](#).

### 1.2.5 Is it reasonable to propose incompatible changes to Python?

In general, no. There are already millions of lines of Python code around the world, so any change in the language that invalidates more than a very small fraction of existing programs has to be frowned upon. Even if you can provide a conversion program, there's still the problem of updating all documentation; many books have been written about Python, and we don't want to invalidate them all at a single stroke.

Providing a gradual upgrade path is necessary if a feature has to be changed. [PEP 5](#) describes the procedure followed for introducing backward-incompatible changes while minimizing disruption for users.

### 1.2.6 Is Python a good language for beginning programmers?

Yes.

It is still common to start students with a procedural and statically typed language such as Pascal, C, or a subset of C++ or Java. Students may be better served by learning Python as their first language. Python has a very simple and consistent syntax and a large standard library and, most importantly, using Python in a beginning programming course lets students concentrate on important programming skills such as problem decomposition and data type design. With Python, students can be quickly introduced to basic concepts such as loops and procedures. They can probably even work with user-defined objects in their very first course.

For a student who has never programmed before, using a statically typed language seems unnatural. It presents additional complexity that the student must master and slows the pace of the course. The students are trying to learn to think like a computer, decompose problems, design consistent interfaces, and encapsulate data. While learning to use a statically typed language is important in the long term, it is not necessarily the best topic to address in the students' first programming course.

Many other aspects of Python make it a good first language. Like Java, Python has a large standard library so that students can be assigned programming projects very early in the course that *do* something. Assignments aren't restricted to the standard four-function calculator and check balancing programs. By using the standard library, students can gain the satisfaction of working on realistic applications as they learn the fundamentals of programming. Using the standard library also teaches students about code reuse. Third-party modules such as PyGame are also helpful in extending the students' reach.

Python's interactive interpreter enables students to test language features while they're programming. They can keep a window with the interpreter running while they enter their program's source in another window. If they can't remember the methods for a list, they can do something like this:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
['__dir__', '__doc__', '__eq__', '__format__', '__ge__',
['__getattr__', '__getitem__', '__gt__', '__hash__', '__iadd__',
['__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
['__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
['__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
['__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
↪ 'reverse', 'sort']

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end

>>> L.append(1)
>>> L
[1]
```

With the interpreter, documentation is never far from the student as they are programming.

There are also good IDEs for Python. IDLE is a cross-platform IDE for Python that is written in Python using Tkinter. Emacs users will be happy to know that there is a very good Python mode for Emacs. All of these programming environments provide syntax highlighting, auto-indenting, and access to the interactive interpreter while coding. Consult [the Python wiki](#) for a full list of Python editing environments.

If you want to discuss Python's use in education, you may be interested in joining [the edu-sig mailing list](#).





## 2.1 Γενικές Ερωτήσεις

### 2.1.1 Υπάρχει πρόγραμμα εντοπισμού σφαλμάτων σε επίπεδο πηγαίου κώδικα με σημεία διακοπής , με ένα βήμα κλπ.;

Ναι.

Πολλοί εντοπιστές σφαλμάτων για την Python περιγράφονται παρακάτω και η ενσωματωμένη συνάρτηση `breakpoint()` σάς επιτρέπει να μεταβείτε σε οποιοδήποτε από αυτά.

Το `pdb` module είναι ένα απλό αλλά επαρκές πρόγραμμα εντοπισμού σφαλμάτων σε λειτουργία κονσόλας για την Python. Είναι μέρος της τυπικής βιβλιοθήκης Python και είναι *documented in the Library Reference Manual*. Μπορείτε επίσης να γράψετε το δικό σας πρόγραμμα σφαλμάτων χρησιμοποιώντας τον κώδικα για το `pdb` ως παράδειγμα.

Το διαδραστικό περιβάλλον ανάπτυξης IDLE, το οποίο αποτελεί μέρος της τυπικής διανομής Python (συνήθως διαθέσιμο ως [Tools/scripts/idle3](#)), που περιλαμβάνει ένα γραφικό πρόγραμμα εντοπισμού σφαλμάτων.

Το PythonWin είναι ένα Python IDE που περιλαμβάνει ένα πρόγραμμα εντοπισμού σφαλμάτων GUI που βασίζεται σε `pdb`. Το πρόγραμμα εντοπισμού σφαλμάτων PythonWin χρωματίζει τα σημεία διακοπής και έχει αρκετά ωραία χαρακτηριστικά, όπως τον εντοπισμό σφαλμάτων σε προγράμματα που δεν είναι PythonWin. Το PythonWin είναι διαθέσιμο ως μέρος του [pywin32](#) έργου και ως μέρος της διανομής [ActivePython](#).

Το [Eric](#) είναι ένα IDE που βασίζεται στο PyQt και το component επεξεργασίας Scintilla.

Το [trepan3k](#) είναι ένα πρόγραμμα εντοπισμού σφαλμάτων παρόμοιο με το `gdb`.

Το [Visual Studio Code](#) είναι ένας IDE με εργαλεία εντοπισμού σφαλμάτων που ενσωματώνεται με λογισμικό ελέγχου έκδοσης.

Υπάρχει ένας αριθμός εμπορικών Python IDEs που περιλαμβάνουν γραφικούς εντοπισμούς σφαλμάτων. Αυτά περιλαμβάνουν:

- [Wing IDE](#)
- [Komodo IDE](#)

- [PyCharm](#)

## 2.1.2 Υπάρχουν εργαλεία που βοηθούν στην εύρεση σφαλμάτων ή στην εκτέλεση στατικής ανάλυσης;

Ναι.

[Pylint](#) και [Pyflakes](#) κάνουν βασικό έλεγχο που θα σας βοηθήσει να εντοπίσετε τα σφάλματα νωρίτερα.

Έλεγχοι στατικού τύπου όπως [Mypy](#), [Pyre](#), και [Pytype](#) μπορεί να ελέγξει υποδείξεις τύπου στον πηγαίο κώδικα της Python.

## 2.1.3 Πως μπορώ να δημιουργήσω ένα stand-alone binary από ένα Python script;

Δεν χρειάζεστε την δυνατότητα μεταγλώττισης κώδικα Python σε C, εάν το μόνο που θέλετε είναι ένα stand-alone πρόγραμμα που οι χρήστες μπορούν να κατεβάσουν και να εκτελέσουν χωρίς να χρειάζεται να εγκαταστήσουν πρώτα την διανομή της Python. Υπάρχει μια σειρά από εργαλεία που καθορίζουν το σύνολο των modules που απαιτούνται από ένα πρόγραμμα και συνδέουν αυτά τα modules μαζί με ένα δυαδικό αρχείο Python για να παχθεί ένα μόνο εκτελέσιμο.

Το ένα είναι να χρησιμοποιήσετε το εργαλείο παγώματος, το οποίο περιλαμβάνεται στο δέντρο πηγαίου της Python ως [Tools/freeze](#). Μετατρέπει δυαδικό κώδικα Python σε C πίνακες• με έναν C μεταγλωττιστή μπορείτε να ενσωματώσετε όλα τα modules σας σε ένα νέο πρόγραμμα, το οποίο στη συνέχεια συνδέεται με τα τυπικά Python modules.

Λειτουργεί σαρώνοντας τον πηγαίο κώδικα αναδρομικά για δηλώσεις εισαγωγής (και στις δύο μορφές) και αναζητώντας τα modules στην τυπική διαδρομή Python καθώς και στον κατάλογο προέλευσης (για ενσωματωμένα modules). Στην συνέχεια γυρίζει τα bytecode για modules που είναι γραμμένες σε Python σε κώδικα C (initializers πίνακα που μπορούν να μετατραπούν σε αντικείμενα κώδικα χρησιμοποιώντας το marshal module) και δημιουργεί ένα προσαρμοσμένο αρχείο διαμόρφωσης που περιέχει μόνο εκείνα τα ενσωματωμένα modules που χρησιμοποιούνται πραγματικά στο πρόγραμμα και το συνδέει με τον υπόλοιπο διερμηνέα Python για να σχηματίσει ένα αυτόνομο δυαδικό αρχείο που λειτουργεί ακριβώς όπως το σενάριο σας.

Τα παρακάτω πακέτα μπορούν να σας βοηθήσουν για την δημιουργία εκτελέσιμων κονσόλας και GUI:

- [Nuitka](#) (Cross-platform)
- [PyInstaller](#) (Cross-platform)
- [PyOxidizer](#) (Cross-platform)
- [cx\\_Freeze](#) (Cross-platform)
- [py2app](#) (macOS μόνο)
- [py2exe](#) (Windows μόνο)

## 2.1.4 Υπάρχουν πρότυπα κωδικοποίησης ή οδηγός στυλ για προγράμματα Python;

Ναι. Το στυλ κωδικοποίησης που απαιτείται για τα τυπικά modules βιβλιοθήκης τεκμηριώνεται ως [PEP 8](#).

## 2.2 Βασική Γλώσσα

### 2.2.1 Γιατί λαμβάνω ένα UnboundLocalError όταν η μεταβλητή έχει μια τιμή;

Μπορεί να είναι έκπληξη να λάβετε το `UnboundLocalError` σε κώδικα που λειτουργούσε προηγουμένως όταν τροποποιείται προσθέτοντας μια δήλωση εκχώρησης κάπου στο σώμα μιας συνάρτησης.

Αυτό ο κώδικας:

```
>>> x = 10
>>> def bar():
...     print(x)
...
>>> bar()
10
```

δουλεύει, αλλά αυτός ο κώδικας:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

καταλήγει σε ένα `UnboundLocalError`:

```
>>> foo()
Traceback (most recent call last):
...
UnboundLocalError: local variable 'x' referenced before assignment
```

Αυτό συμβαίνει επειδή όταν κάνετε μια ανάθεση σε μια μεταβλητή σε ένα εύρος, αυτή η μεταβλητή γίνεται τοπική σε αυτό το εύρος και σκιάζει οποιαδήποτε μεταβλητή με παρόμοιο όνομα στο εξωτερικό εύρος. Εφόσον η τελευταία πρόταση στο `foo` εκχωρεί μια νέα τιμή στο `x`, ο μεταγλωττιστής την αναγνωρίζει ως τοπική μεταβλητή. Κατά συνέπεια όταν η προηγούμενη `print(x)` επιχειρεί να εκτυπώσει την uninitialized τοπική μεταβλητή και προκύπτει σφάλμα.

Στο παραπάνω παράδειγμα, μπορείτε να αποκτήσετε πρόσβαση στη μεταβλητή εξωτερικού εύρους δηλώνοντας την ως καθολική:

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
...
>>> foobar()
10
```

Αυτή η ρητή δήλωση απαιτείται για να σας υπενθυμίσει ότι (σε αντίθεση με την επιφανειακά ανάλογη κατάσταση με τις μεταβλητές κλάσης και στιγμιότυπου) στην πραγματικότητα τροποποιείτε την τιμή της μεταβλητής στο εξωτερικό πεδίο:

```
>>> print(x)
11
```

Μπορείτε να κάνετε κάτι παρόμοιο σε ένα ένθετο πεδίο χρησιμοποιώντας τη λέξη κλειδί `nonlocal`:

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
...
>>> foo()
10
11
```

## 2.2.2 Ποιοι είναι οι κανόνες για τις τοπικές και τις καθολικές μεταβλητές στην Python;

Στην Python, οι μεταβλητές που αναφέρονται μόνο μέσα σε μια συνάρτηση είναι έμμεσα καθολικές. Εάν σε μια μεταβλητή εκχωρηθεί μια τιμή οπουδήποτε μέσα στο σώμα της συνάρτησης, θεωρείται ότι είναι τοπική, εκτός αν δηλωθεί ρητά ως καθολική.

Αν και είναι λίγο έκπληξη στην αρχή, αυτό το εξηγεί μια στιγμή. Από τη μία πλευρά η απαίτηση `global` για εκχωρημένες μεταβλητές παρέχει μια γραμμή έναντι ανεπιθύμητων παρενεργειών. Από την άλλη, εάν απαιτείται `global` για όλες τις καθολικές αναφορές, θα χρησιμοποιούσατε `global` όλη την ώρα. Θα έπρεπε να δηλώσετε ως καθολική κάθε αναφορά σε μια ενσωματωμένη λειτουργία ή σε ένα στοιχείο ενός εισαγόμενου `module`. Αυτή η ακαταστασία θα νικήσει την χρησιμότητα της δήλωσης `global` για τον εντοπισμό παρενεργειών.

## 2.2.3 Γιατί τα lambdas που ορίζονται σε έναν βρόχο με διαφορετικές τιμές επιστρέφουν όλα το ίδιο αποτέλεσμα;

Ας υποθέσουμε ότι χρησιμοποιείτε έναν βρόχο `for` για να ορίσετε μερικά διαφορετικά lambdas (ή ακόμα και απλές συναρτήσεις), π.χ.:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

Αυτό σας δίνει μια λίστα που περιέχει 5 lambdas που υπολογίζουν το `x**2`. Μπορεί να περιμένετε ότι, όταν καλέσετε, θα επέστρεφαν, αντίστοιχα, 0, 1, 4, 9 και 16. Ωστόσο, όταν δοκιμάσετε πραγματικά θα δείτε ότι όλα επιστρέφουν 16:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

Αυτό συμβαίνει επειδή το `x` δεν είναι τοπικό για τα lambdas, αλλά ορίζεται στο εξωτερικό εύρος και είναι προ-σβάσιμο όταν το lambda καλείται — όχι όταν ορίζεται. Στο τέλος του βρόχου, η τιμή του `x` είναι 4, επομένως όλες οι συναρτήσεις επιστρέφουν τώρα `4**2`, δηλαδή 16. Μπορείτε επίσης να το επαληθεύσετε αλλάζοντας την τιμή του `x` και δείτε πως αλλάζουν τα αποτελέσματα του lambda:

```
>>> x = 8
>>> squares[2] ()
64
```

Για να το αποφύγετε αυτό, πρέπει να αποθηκεύσετε τις τιμές σε μεταβλητές τοπικές στο lambda, έτσι ώστε να μην βασίζονται στην τιμή του καθολικού x:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Εδώ, το n=x δημιουργεί μια νέα μεταβλητή n τοπική στο lambda και υπολογίζεται όταν το lambda ορίζεται έτσι ώστε να έχει την ίδια τιμή που είχε το x σε εκείνο το σημείο βρόχου. Αυτό σημαίνει ότι η τιμή του n θα είναι 0 στο πρώτο lambda, 1 στο δεύτερο, 2 στο τρίτο και ούτω καθεξής. Επομένως κάθε lambda θα επιστρέψει τώρα το σωστό αποτέλεσμα:

```
>>> squares[2] ()
4
>>> squares[4] ()
16
```

Σημειώστε ότι αυτή η συμπεριφορά δεν είναι ιδιόμορφη για το lambdas, αλλά ισχύει και για κανονικές λειτουργίες.

## 2.2.4 Πως μοιράζομαι καθολικές μεταβλητές σε modules;

Ο κανονικός τρόπος για να μοιράξετε πληροφορίες μεταξύ των λειτουργικών μονάδων μέσα σε ένα μόνο πρόγραμμα είναι η δημιουργία ενός ειδικού module (συνχνά ονομάζεται config ή cfg). Απλώς εισαγάγετε το module διαμόρφωσης σε όλες τα modules της εφαρμογής σας• το module στην συνέχεια γίνεται διαθέσιμο ως παγκόσμιο όνομα. Επειδή υπάρχει μόνο ένα παράδειγμα για κάθε module, οι αλλαγές που γίνονται στο αντικείμενο του module αντικατοπτρίζονται παντού. Για παράδειγμα:

config.py:

```
x = 0    # Default value of the 'x' configuration setting
```

mod.py:

```
import config
config.x = 1
```

main.py:

```
import config
import mod
print(config.x)
```

Λάβετε υπόψη ότι η χρήση ενός module είναι επίσης η βάση για την εφαρμογή του μοτίβου σχεδιασμού singleton, για τον ίδιο λόγο.

## 2.2.5 Ποιες είναι οι «βέλτιστες πρακτικές» για τη χρήση import σε ένα module;

Γενικά, μην χρησιμοποιείτε `from modulename import *`. Κάτι τέτοιο δημιουργεί μια ακαταστασία στο `importer's namespace`, και καθιστά πιο δύσκολο για τα linters να εντοπίσουν απροσδιόριστα ονόματα.

Εισαγωγή modules στην κορυφή ενός αρχείου. Με αυτόν τον τρόπο καθιστά σαφές ποια άλλα modules απαιτεί ο κώδικας σας και αποφεύγονται ερωτήσεις σχετικά με το αν το όνομα της μονάδας είναι εντός πεδίου. Χρησιμοποιώντας ένα `import` ανά γραμμή καθιστά εύκολη την προσθήκη και τη διαγραφή `module imports`, αλλά χρησιμοποιώντας πολλαπλά `imports` ανά γραμμής καταναλώνεται λιγότερος χώρος στην οθόνη.

Είναι καλή πρακτική εάν εισάγετε module με την ακόλουθη σειρά:

1. τυπικά module βιβλιοθήκης – π.χ. `sys`, `os`, `argparse`, `re`
2. module βιβλιοθήκης τρίτων (ό, τι είναι εγκατεστημένο στο κατάλογο `site-packages` της Python) – π.χ. `dateutil`, `requests`, `PIL.Image`
3. τοπικά αναπτυγμένα modules

Μερικές φορές είναι απαραίτητο να μετακινηθούν οι εισαγωγές σε μια συνάρτηση ή κλάση για να αποφευχθούν προβλήματα με τις κυκλικές εισαγωγές. Ο Gordon McMillan λέει:

Οι κυκλικές εισαγωγές είναι καλές όταν και τα δύο modules χρησιμοποιούν τη μορφή εισαγωγής «`import <module>`». Αποτυγχάνουν όταν το 2ο module θέλει να πάρει ένα όνομα από το πρώτο («`from module import name`») και η εισαγωγή είναι στο κορυφαίο επίπεδο. Αυτό συμβαίνει επειδή το πρώτο module είναι απασχολημένο με την εισαγωγή του 2ου.

Σε αυτήν την περίπτωση, εάν το δεύτερο module χρησιμοποιείται μόνο σε μια συνάρτηση, τότε η εισαγωγή μπορεί εύκολα να μεταφερθεί μέσα σε αυτήν την συνάρτηση. Από τη στιγμή που καλείται η εισαγωγή, το πρώτο module θα έχει ολοκληρώσει την αρχικοποίηση, και το δεύτερο module μπορεί να κάνει την εισαγωγή του.

Μπορεί επίσης να είναι απαραίτητο να μετακινήσετε τις εισαγωγές από το ανώτερο επίπεδο κώδικα, εάν ορισμένα από τα module είναι συγκεκριμένα για την πλατφόρμα. Σε αυτήν την περίπτωση, ενδέχεται να μην είναι καν δυνατή η εισαγωγή όλων των modules στο επάνω μέρος του αρχείου. Σε αυτήν την περίπτωση, η εισαγωγή των σωστών modules στον αντίστοιχο κώδικα για συγκεκριμένη πλατφόρμα είναι μια καλή επιλογή.

Μετακινήστε τις εισαγωγές σε τοπικό πεδίο, όπως μέσα σε έναν ορισμό συνάρτησης, μόνο εάν είναι απαραίτητο να λυθεί ένα πρόβλημα όπως η αποφυγή μιας κυκλικής εισαγωγής ή εάν προσπαθείτε να μειώσετε τον χρόνο προετοιμασίας μιας μονάδας. Αυτή η τεχνική είναι ιδιαίτερα χρήσιμη εάν πολλές από τις εισαγωγές δεν είναι απαραίτητες ανάλογα με τον τρόπο εκτέλεσης του προγράμματος. Μπορείτε επίσης να θέλετε να μετακινήσετε τις εισαγωγές σε μια συνάρτηση εάν τα modules χρησιμοποιούνται μόνο σε αυτήν τη συνάρτηση. Λάβετε υπόψη ότι η φόρτωση ενός module την πρώτη φορά μπορεί να είναι δαπανηρή λόγω της μιας φοράς της αρχικοποίησης του module, αλλά η φόρτωση ενός module πολλές φορές είναι σχεδόν δωρεάν, κοστίζοντας μόνο μερικές αναζητήσεις σε λεξικό. Ακόμα και αν το όνομα του module έχει ξεφύγει από το πεδίο εφαρμογής του, το module είναι πιθανώς διαθέσιμο στο `sys.modules`.

## 2.2.6 Γιατί μοιράζονται οι προεπιλεγμένες τιμές μεταξύ των αντικειμένων;

Αυτός ο τύπος σφάλματος συνήθως δαγκώνει νεοφυείς προγραμματιστές. Σκεφτείτε αυτήν τη συνάρτηση:

```
def foo(mydict={}): # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

Την πρώτη φορά που καλείτε αυτήν την συνάρτηση, το `mydict` περιέχει ένα μεμονωμένο στοιχείο. Τη δεύτερη φορά, το `mydict` περιέχει δύο στοιχεία γιατί όταν το `foo()` ξεκινά την εκτέλεση, το `mydict` ξεκινά με ένα αντικείμενο ήδη μέσα.

Συχνά αναμένεται ότι μια κλήση συνάρτησης δημιουργεί νέα αντικείμενα για προεπιλεγμένες τιμές. Δεν συμβαίνει αυτό. Οι προεπιλεγμένες τιμές δημιουργούνται ακριβώς μία φορά, όταν ορίζεται η συνάρτηση. Εάν αυτό το αντικείμενο αλλάξει, όπως το λεξικό σε αυτό το παράδειγμα, επόμενες κλήσεις στην συνάρτηση θα αναφέρονται σε αυτό το αλλαγμένο αντικείμενο.

Εξ ορισμού, αμετάβλητα αντικείμενα όπως αριθμοί, συμβολοσειρές, πλειάδες και `None`, είναι ασφαλή από αλλαγές. Οι αλλαγές σε μεταβλητά αντικείμενα όπως λεξικά, λίστες και παρουσίες κλάσεων μπορεί να οδηγήσουν σε σύγχυση.

Λόγω αυτής της δυνατότητας, είναι καλή πρακτική προγραμματισμού να μην χρησιμοποιείτε μεταβλητά αντικείμενα ως προεπιλεγμένες τιμές. Αντίθετα, χρησιμοποιείτε `None` ως προεπιλεγμένη τιμή και μέσα στην συνάρτηση, ελέγξτε εάν η παράμετρος είναι `None` και δημιουργήστε μια νέα λίστα/λεξικά/ό,τι και αν είναι. Για παράδειγμα μην γράψετε:

```
def foo(mydict={}):
    ...
```

αλλά:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {} # create a new dict for local namespace
```

Αυτή η δυνατότητα μπορεί να είναι χρήσιμη. Όταν έχετε μια συνάρτηση που είναι χρονοβόρα για τον υπολογισμό, μια κοινή τεχνική είναι να αποθηκεύσετε προσωρινά τις παραμέτρους και την επιστρεφόμενη τιμή κάθε κλήσης στη συνάρτηση και να επιστρέψετε την προσωρινά αποθηκευμένη τιμή εάν ζητηθεί ξανά η ίδια τιμή. Αυτό ονομάζεται «memoizing», και μπορεί να εφαρμοστεί ως εξής:

```
# Callers can only provide two parameters and optionally pass _cache by keyword
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result # Store result in the cache
    return result
```

Θα μπορούσατε να χρησιμοποιήσετε μια καθολική μεταβλητή που περιέχει ένα λεξικό αντί για την προεπιλεγμένη τιμή. Είναι θέμα γούστου.

## 2.2.7 Πώς μπορώ να μεταβιβάσω προαιρετικές παραμέτρους ή παραμέτρους λέξεων-κλειδιών από τη μια συνάρτηση στην άλλη;

Συλλέξτε τα ορίσματα χρησιμοποιώντας τους specifiers `*` και `**` στη λίστα παραμέτρων της συνάρτησης. Αυτό σας δίνει τα ορίσματα θέσης ως πλειάδα και τα ορίσματα λέξεων-κλειδιών ως λεξικό. Στη συνέχεια, μπορείτε να μεταβιβάσετε αυτά τα ορίσματα κατά την κλήση άλλης συνάρτησης με τη χρήση των `*` και `**`:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

## 2.2.8 Ποια είναι η διαφορά μεταξύ ορισμάτων και παραμέτρων;

*Παράμετροι* ορίζονται από τα ονόματα που εμφανίζονται στον ορισμό μιας συνάρτησης, ενώ τα *ορίσματα* είναι οι τιμές που πραγματικά μεταβιβάζονται σε μια συνάρτηση κατά την κλήση της. Οι παράμετροι ορίζουν τι *kind of arguments* μπορεί να δεχτεί μια συνάρτηση. Για παράδειγμα δεδομένου του ορισμού της συνάρτησης:

```
def func(foo, bar=None, **kwargs):
    pass
```

*foo*, *bar* και *kwargs* είναι παράμετροι της *func*. Ωστόσο, όταν καλείται η *func*, για παράδειγμα:

```
func(42, bar=314, extra=somevar)
```

οι τιμές 42, 314, και *somevar* είναι ορίσματα.

## 2.2.9 Γιατί η αλλαγή της λίστας “y” αλλάζει επίσης και τη λίστα “x”;

Αν γράψατε κώδικα όπως:

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

μπορεί να αναρωτιέστε γιατί η προσθήκη ενός στοιχείου στο *y* άλλαξε επίσης το *x*.

Υπάρχουν δύο παράγοντες που παράγουν αυτό το αποτέλεσμα:

- 1) Οι μεταβλητές είναι απλά ονόματα που αναφέρονται σε αντικείμενα. Κανόνας  $y = x$  δεν δημιουργείται αντίγραφο της λίστας – δημιουργεί μια νέα μεταβλητή *y* που αναφέρεται στο ίδιο αντικείμενο που αναφέρεται το *x*. Αυτό σημαίνει ότι υπάρχει μόνο ένα αντικείμενο (η λίστα), και το *x* και το *y* αναφέρονται σε αυτό.
- 2) Οι λίστες είναι *mutable*, που σημαίνει ότι μπορείτε να αλλάξετε το περιεχόμενό τους.

Μετά την κλήση στο `append()`, το περιεχόμενο του μεταβλητού αντικειμένου έχει αλλάξει από `[]` σε `[10]`. Επειδή και οι δύο μεταβλητές αναφέρονται στο ίδιο αντικείμενο, χρησιμοποιώντας οποιοδήποτε όνομα αποκτά πρόσβαση στην τροποποιημένη τιμή `[10]`.

Αν αντιστοιχίσουμε ένα αμετάβλητο αντικείμενο σε *x*:

```
>>> x = 5 # ints are immutable
>>> y = x
>>> x = x + 1 # 5 can't be mutated, we are creating a new object here
>>> x
6
>>> y
5
```

μπορούμε να δούμε ότι σε αυτήν την περίπτωση *x* και *y* δεν είναι πλέον ίσα. Αυτό συμβαίνει επειδή οι ακέραιοι αριθμοί είναι *immutable*, και όταν κάνουμε  $x = x + 1$  δεν μεταλλάσσεται το `int 5` αυξάνοντας την τιμή του· αντίθετα, δημιουργούμε ένα νέο αντικείμενο (το `int 6`) και το εκχωρούμε στο *x* (δηλαδή, αλλάζοντας το αντικείμενο στο οποίο αναφέρεται το *x*). Μετά από αυτήν την ανάθεση έχουμε δύο αντικείμενα (τα `ints 6 and 5`) και δύο μεταβλητές που αναφέρονται σε αυτά (*x* τώρα αναφέρεται σε 6 αλλά *y* ακόμα αναφέρεται στο 5).



Ορισμένες λειτουργίες (για παράδειγμα `y.append(10)` και `y.sort()`) μεταλλάσσουν το αντικείμενο, ενώ επιφανειακά παρόμοιες πράξεις (για παράδειγμα `y = y + [10]` και `sorted(y)`) δημιουργούν ένα νέο αντικείμενο. Γενικά στην Python (και σε όλες τις περιπτώσεις στην τυπική βιβλιοθήκη) μια μέθοδος που μεταλλάσσει ένα αντικείμενο θα επιστρέψει `None` για να αποφύγει τη σύγχυση των δύο τύπων πράξεων. Επομένως, εάν γράψετε κατά λάθος `y.sort()` νομίζοντας ότι θα σας δώσει ένα ταξινομημένο αντίγραφο του `y`, θα καταλήξετε με `None`, το οποίο πιθανότατα θα προκαλέσει το πρόγραμμά σας να δημιουργήσει ένα λάθος το οποίο διαγιγνώσκεται εύκολα.

Ωστόσο, υπάρχει μία κατηγορία λειτουργιών όπου η ίδια πράξη έχει μερικές φορές διαφορετικές συμπεριφορές με διαφορετικούς τύπους: οι επαυξημένοι τελεστές ανάθεσης. Για παράδειγμα, `+=` μεταλλάσσει τις λίστες αλλά όχι τις πλειάδες ή τους `ints` (`a_list += [1, 2, 3]` είναι ίσο με `a_list.extend([1, 2, 3])` και μεταλλάσσει `a_list`, ενώ το `some_tuple += (1, 2, 3)` και `some_int += 1` δημιουργεί νέα αντικείμενα).

Με άλλα λόγια:

- Εάν έχουμε ένα μεταβλητό αντικείμενο (`list`, `dict`, `set`, κλπ.), μπορούμε να χρησιμοποιήσουμε ορισμένες συγκεκριμένες λειτουργίες για να το μεταλλάξουμε και όλες τις μεταβλητές που αναφέρονται σε αυτό θα δουν την αλλαγή.
- Εάν έχουμε ένα αμετάβλητο αντικείμενο (`str`, `int`, `tuple`, κλπ.), όλες οι μεταβλητές που αναφέρονται σε αυτό θα βλέπουν πάντα την ίδια τιμή, αλλά οι λειτουργίες που θα μετατρέπουν αυτήν την τιμή σε μια νέα τιμή επιστρέφουν πάντα ένα νέο αντικείμενο.

Εάν θέλετε να γνωρίζετε εάν δύο μεταβλητές αναφέρονται στο ίδιο αντικείμενο ή όχι, μπορείτε να χρησιμοποιήσετε τον `is` operator, ή την ενσωματωμένη συνάρτηση `id()`.

## 2.2.10 Πως μπορώ να γράψω μια συνάρτηση με παραμέτρους εξόδου (κλήση με αναφορά);

Να θυμάστε ότι τα ορίσματα μεταβιβάζονται με ανάθεση στην Python. Εφόσον η εκχώρηση δημιουργεί απλώς αναφορές σε αντικείμενα, δεν υπάρχει ψευδώνυμο μεταξύ ενός ονόματος ορίσματος σε αυτό που καλεί και σε αυτό που καλείται, και επομένως δεν υπάρχει κλήση προς αναφορά από μόνη της. Μπορείτε να επιτύχετε το επιθυμητό αποτέλεσμα με διάφορους τρόπους.

- 1) Επιστρέφοντας μια πλειάδα των αποτελεσμάτων:

```
>>> def func1(a, b):
...     a = 'new-value'           # a and b are local names
...     b = b + 1                 # assigned to new objects
...     return a, b              # return new values
...
>>> x, y = 'old-value', 99
>>> func1(x, y)
('new-value', 100)
```

Αυτή είναι σχεδόν πάντα η πιο ξεκάθαρη λύση.

- 2) Χρησιμοποιώντας καθολικές μεταβλητές. Αυτό δεν είναι ασφαλές για νήμα και δεν συνίσταται.
- 3) Περνώντας ένα μεταβλητό (με δυνατότητα αλλαγής επί τόπου) αντικείμενο:

```
>>> def func2(a):
...     a[0] = 'new-value'       # 'a' references a mutable list
...     a[1] = a[1] + 1          # changes a shared object
...
>>> args = ['old-value', 99]
>>> func2(args)
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
>>> args
['new-value', 100]
```

4) Περνώντας σε ένα λεξικό που μεταλλάσσεται:

```
>>> def func3(args):
...     args['a'] = 'new-value'      # args is a mutable dictionary
...     args['b'] = args['b'] + 1    # change it in-place
...
>>> args = {'a': 'old-value', 'b': 99}
>>> func3(args)
>>> args
{'a': 'new-value', 'b': 100}
```

5) Ή ομαδοποιήστε τιμές σε μια παρουσία κλάσης:

```
>>> class Namespace:
...     def __init__(self, /, **args):
...         for key, value in args.items():
...             setattr(self, key, value)
...
>>> def func4(args):
...     args.a = 'new-value'          # args is a mutable Namespace
...     args.b = args.b + 1           # change object in-place
...
>>> args = Namespace(a='old-value', b=99)
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

Δεν υπάρχει σχεδόν ποτέ καλός λόγος να γίνει αυτό περίπλοκο.

Η καλύτερη επιλογή σας είναι να επιστρέψετε μια πλειάδα που περιέχει πολλαπλά αποτελέσματα.

## 2.2.11 Πως δημιουργείτε μια συνάρτηση υψηλότερης τάξης στην Python;

Έχετε δύο επιλογές: μπορείτε να χρησιμοποιήσετε ένθετα πεδία ή μπορείτε να χρησιμοποιήσετε callable αντικείμενα. Για παράδειγμα, ας υποθέσουμε ότι θέλετε να ορίσετε το `linear(a, b)` που επιστρέφει μια συνάρτηση `f(x)` που υπολογίζει την τιμή  $a \cdot x + b$ . Χρησιμοποιώντας ένθετα πεδία:

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

Ή χρησιμοποιώντας ένα callable αντικείμενο:

```
class linear:
    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a * x + self.b
```

Και στις δύο περιπτώσεις:

```
taxes = linear(0.3, 2)
```

δίνει ένα callable αντικείμενο όπου `taxes(10e6) == 0.3 * 10e6 + 2`.

Η προσέγγιση του callable αντικειμένου έχει το μειονέκτημα ότι είναι λίγο πιο αργή και οδηγεί σε ελαφρώς μεγαλύτερο κώδικα. Ωστόσο, σημειώστε ότι μια συλλογή από callables μπορεί να μοιραστεί την υπογραφή τους μέσω κληρονομικότητας:

```
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

Το αντικείμενο μπορεί να ενθυλακώσει την κατάσταση για πολλές μεθόδους:

```
class counter:

    value = 0

    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

Εδώ `inc()`, `dec()` και `reset()` λειτουργούν σαν συναρτήσεις που μοιράζονται την ίδια μεταβλητή μέτρησης.

## 2.2.12 Πως μπορώ να αντιγράψω ένα αντικείμενο στην Python;

Γενικά, δοκιμάστε `copy.copy()` ή `copy.deepcopy()` για τη γενική περίπτωση. Δεν μπορούν να αντιγραφούν όλα τα αντικείμενα, αλλά τα περισσότερα μπορούν.

Ορισμένα αντικείμενα μπορούν να αντιγραφούν πιο εύκολα. Τα λεξικά έχουν μία μέθοδο `copy()`:

```
newdict = olddict.copy()
```

Οι ακολουθίες μπορούν να αντιγραφούν με τεμαχισμό:

```
new_l = l[:]
```

### 2.2.13 Πως μπορώ να βρω τις μεθόδους ή τα χαρακτηριστικά ενός αντικειμένου;

Για παράδειγμα μια κλάση `x` που ορίζεται από το χρήστη, `dir(x)` επιστρέφει μια αλφαβητική λίστα με τα ονόματα που περιέχει τα χαρακτηριστικά της οντότητας και τις μεθόδους και τα χαρακτηριστικά που ορίζονται από την κλάση του.

### 2.2.14 Πως μπορεί ο κώδικας μου να ανακαλύψει το όνομα ενός αντικειμένου;

Γενικά μιλώντας, δεν μπορεί, επειδή τα αντικείμενα δεν έχουν πραγματικά ονόματα. Ουσιαστικά, η εκχώρηση δεσμεύει πάντα ένα όνομα σε μια τιμή• το ίδιο ισχύει για τις δηλώσεις `def` και `class`, αλλά σε αυτή την περίπτωση η τιμή είναι callable. Λάβετε υπόψη τον ακόλουθο κώδικα:

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print(b)
<__main__.A object at 0x16D07CC>
>>> print(a)
<__main__.A object at 0x16D07CC>
```

Αναμφισβήτητα η κλάση έχει ένα όνομα: παρόλο που είναι δεσμευμένη σε δύο ονόματα και καλείται μέσω του ονόματος `B` η δημιουργημένη οντότητα εξακολουθεί να αναφέρεται ως οντότητα της κλάσης `A`. Ωστόσο, είναι αδύνατο να πούμε είτε το όνομα της οντότητας είναι `a` ή `b`, καθώς και τα δύο ονόματα συνδέονται με την ίδια τιμή.

Γενικά μιλώντας, δεν θα πρέπει να είναι απαραίτητο ο κώδικας σας να «γνωρίζει τα ονόματα» συγκεκριμένων τιμών. Αν δεν γράφετε εσκευμένα ενδοσκοπικά (introspective) προγράμματα, αυτό είναι συνήθως μια ένδειξη ότι μια αλλαγή προσέγγισης μπορεί να είναι επωφέλης.

Στο `comp.lang.python`, ο Fredrik Lundh έδωσε μια εξαιρετική αναλογία ως απάντηση σε αυτήν την ερώτηση:

Με το ίδιο τρόπο που παίρνετε το όνομα αυτής της γάτας που βρήκατε στη βεράντα σας: ή ίδια γάτα (αντικείμενο) δεν μπορεί να σας πει το όνομά της και δεν την ενδιαφέρει πραγματικά - έτσι ο μόνος τρόπος για να μάθετε πως λέγεται είναι να ρωτήσετε όλους τους γείτονές σας (namespaces) αν είναι η γάτα τους (αντικείμενο)...

....και μην εκπλαγείτε αν διαπιστώσετε ότι είναι γνωστό με πολλά ονόματα, ή κανένα όνομα!

### 2.2.15 Τι συμβαίνει με την προτεραιότητα του τελεστή κόμματος;

Το κόμμα δεν είναι τελεστής στην Python. Σκεφτείτε αυτήν την συνεδρία:

```
>>> "a" in "b", "a"
(False, 'a')
```

Δεδομένου ότι το κόμμα δεν είναι τελεστής, αλλά διαχωριστικό μεταξύ των εκφράσεων, τα παραπάνω αξιολογούνται σαν να είχατε εισαγάγει:

```
("a" in "b"), "a"
```

δεν:

```
"a" in ("b", "a")
```

Το ίδιο ισχύει για τους διάφορους τελεστές εκχώρησης (=, += κλπ). Δεν είναι πραγματικά τελεστές αλλά συντακτικοί delimiters σε δηλώσεις εκχώρησης.

## 2.2.16 Υπάρχει ισοδύναμο του τριαδικού τελεστή «?:» της C;

Ναι υπάρχει, Η σύνταξη έχει ως εξής:

```
[on_true] if [expression] else [on_false]

x, y = 50, 25
small = x if x < y else y
```

Πριν εισαχθεί αυτή η σύνταξη στην Python 2.5, ένα κοινό ιδίωμα ήταν η χρήση λογικών τελεστών:

```
[expression] and [on_true] or [on_false]
```

Ωστόσο, αυτό το ιδίωμα δεν είναι ασφαλές, καθώς μπορεί να δώσει λανθασμένα αποτελέσματα όταν το *on\_true* έχει ψευδή δυαδική τιμή. Επομένως, είναι πάντα καλύτερο να χρησιμοποιήσετε τη φόρμα ... if ... else ....

## 2.2.17 Είναι δυνατόν να γράψουμε ασαφή one-liners στην Python;

Ναι. Συνήθως αυτό γίνεται με ένθεση του `lambda` εντός του `lambda`. Δείτε τα ακόλουθα τρία παραδείγματα, ελαφρώς προσαρμοσμένα από τον Ulf Bartelt:

```
from functools import reduce

# Primes < 1000
print(list(filter(None, map(lambda y: y*reduce(lambda x, y: x*y!=0,
map(lambda x, y: y% x, range(2, int(pow(y, 0.5)+1))), 1), range(2, 1000)))))

# First 10 Fibonacci numbers
print(list(map(lambda x, f: lambda x, f: (f(x-1, f)+f(x-2, f)) if x>1 else 1:
f(x, f), range(10))))

# Mandelbrot set
print((lambda Ru, Ro, Iu, Io, IM, Sx, Sy: reduce(lambda x, y: x+'\n'+y, map(lambda y,
Iu=Iu, Io=Io, Ru=Ru, Ro=Ro, Sy=Sy, L=lambda yc, Iu=Iu, Io=Io, Ru=Ru, Ro=Ro, i=IM,
Sx=Sx, Sy=Sy: reduce(lambda x, y: x+y, map(lambda x, xc=Ru, yc=yc, Ru=Ru, Ro=Ro,
i=i, Sx=Sx, F=lambda xc, yc, x, y, k, f: lambda xc, yc, x, y, k, f: (k<=0) or (x*x+y*y
>=4.0) or 1+f(xc, yc, x*x-y*y+xc, 2.0*x*y+yc, k-1, f): f(xc, yc, x, y, k, f): chr(
64+F(Ru+x*(Ro-Ru)/Sx, yc, 0, 0, i)), range(Sx)): L(Iu+y*(Io-Iu)/Sy), range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#      \_____/      \_____/      |      |      | lines on screen
#          V          V          |      |_____ columns on screen
#          |          |          |_____ maximum of "iterations"
#          |          |_____ range on y axis
#          |_____ range on x axis
```

Μην το δοκιμάσετε στο σπίτι, παιδιά!

## 2.2.18 Τι σημαίνει η κάθετος(/) στη λίστα παραμέτρων μιας συνάρτησης;

Μια κάθετος στη λίστα ορισμάτων μιας συνάρτησης υποδηλώνει ότι οι παράμετροι πριν από αυτήν είναι μόνο θέσης. Οι παράμετροι μόνο θέσης είναι εκείνες χωρίς εξωτερικά χρησιμοποιήσιμο όνομα. Κατά την κλήση μιας συνάρτησης που δέχεται παραμέτρους μόνο θέσης, τα ορίσματα αντιστοιχίζονται σε παραμέτρους που βασίζονται αποκλειστικά στη θέση τους. Για παράδειγμα η `divmod()` είναι μια συνάρτηση που δέχεται μόνο παραμέτρους θέσης. Η τεκμηρίωσή τους μοιάζει με αυτό:

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y). Invariant: div*y + mod == x.
```

Η κάθετος στο τέλος της λίστας παραμέτρων σημαίνει ότι και οι δύο παράμετροι είναι μόνο θέσης. Επομένως, η κλήση της `divmod()` με ορίσματα λέξεων κλειδιών θα οδηγήσει σε σφάλμα:

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

## 2.3 Αριθμοί και συμβολοσειρές

### 2.3.1 Πώς προσδιορίζω δεκαεξαδικούς ή οκταδικούς ακέραιους αριθμούς;

Για να καθορίσετε έναν οκταδικό ψηφίο, προηγείται την οκταδικής τιμής με ένα μηδέν, "και μετά ένα πεζό ή κεφαλαίο «ο»". Για παράδειγμα, για να ορίσετε τη μεταβλητή «a» στην οκταδική τιμή «10» (8 σε δεκαδικό), πληκτρολογήστε:

```
>>> a = 0o10
>>> a
8
```

Το δεκαεξαδικό είναι εξίσου εύκολο. Απλώς προηγείται του δεκαεξαδικού αριθμού με ένα μηδέν, και μετά ένα πεζό ή κεφαλαίο «x». Τα δεκαεξαδικά ψηφία μπορούν να καθοριστούν με πεζά ή κεφαλαία. Για παράδειγμα, στον διερμηνέα Python:

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

### 2.3.2 Γιατί το `-22 // 10` επιστρέφει `-3`;

Οφείλεται κυρίως στην επιθυμία που το `i % j` να έχει το ίδιο πρόσημο με το `j`. Εάν το θέλετε αυτό, και θέλετε επίσης:

```
i == (i // j) * j + (i % j)
```

τότε η διαίρεση ακεραίου αριθμού πρέπει να επιστρέφει το υπόλοιπο. Η C απαιτεί επίσης να διατηρείται αυτή η ταυτότητα και, στη συνέχεια, οι μεταγλωττιστές που περικλύπτουν το `i // j` πρέπει να κάνουν το `i % j` να έχει το ίδιο πρόσημο με το `i`.

Υπάρχουν λίγες πραγματικές περιπτώσεις χρήσης για το `i % j` όταν το `j` είναι αρνητικό. Όταν το `j` είναι θετικό, υπάρχουν πολλές, και σχεδόν όλες είναι πιο χρήσιμες για `i % j` να είναι  $\geq 0$ . Εάν το ρολόι λέει 10 τι έλεγε πριν από 200 ώρες; `-190 % 12 == 2` είναι χρήσιμο• το `-190 % 12 == -10` είναι ένα σφάλμα που περιμένει να δαγκώσει.

### 2.3.3 Πώς μπορώ να πάρω το literal χαρακτηριστικό `int` αντί για το `SyntaxError`;

Η προσπάθεια αναζήτησης ενός literal χαρακτηριστικού `int` με τον κανονικό τρόπο δίνει ένα `SyntaxError` επειδή η περίοδος θεωρείται ως υποδιαστολή:

```
>>> 1.__class__
File "<stdin>", line 1
  1.__class__
    ^
SyntaxError: invalid decimal literal
```

Η λύση είναι να διαχωριστεί το literal από την τελεία είτε με κενό είτε με παρένθεση.

```
>>> 1 .__class__
<class 'int'>
>>> (1).__class__
<class 'int'>
```

### 2.3.4 Πως μετατρέπω μια συμβολοσειρά σε έναν αριθμό;

For integers, use the built-in `int()` type constructor, e.g. `int('144') == 144`. Similarly, `float()` converts to floating-point, e.g. `float('144') == 144.0`.

Από προεπιλογή, αυτά ερμηνεύουν τον αριθμό ως δεκαδικό, έτσι ώστε `int('0144') == 144` να είναι αληθές, και το `int('0x144')` να εγείρει `ValueError`. Το `int(string, base)` παίρνει τη βάση για μετατροπή ως δεύτερο προαιρετικό όρισμα, οπότε το `int('0x144', 16) == 324`. Εάν η βάση έχει καθοριστεί ως 0, ο αριθμός ερμηνεύεται χρησιμοποιώντας κανόνες της Python: ένα αρχικό "0" υποδηλώνει οκταδικό, και το "0x" δείχνει έναν δεκαεξαδικό αριθμό.

Μην χρησιμοποιείτε την ενσωματωμένη συνάρτηση `eval()` εάν το μόνο που χρειάζεστε είναι να μετατρέψετε συμβολοσειρές σε αριθμούς. `eval()` θα είναι σημαντικά πιο αργή και παρουσιάζει κίνδυνο ασφαλείας: κάποιος θα μπορούσε να σας μεταβιβάσει μια έκφραση Python που μπορεί να έχει ανεπιθύμητες παρενέργειες. Για παράδειγμα κάποιος θα μπορούσε να περάσει το `__import__('os').system("rm -rf $HOME")` το οποίο θα διαγράψει το home φάκελο.

`eval()` έχει επίσης ως αποτέλεσμα την ερμηνεία των αριθμών ως εκφράσεις Python, έτσι ώστε π.χ. το `eval('09')` δίνει ένα συντακτικό σφάλμα επειδή η Python δεν επιτρέπει την εισαγωγή του "0" σε έναν δεκαδικό αριθμό (εκτός "0").

### 2.3.5 Πως μετατρέπω έναν αριθμό σε συμβολοσειρά;

Για να μετατρέψετε, πχ τον αριθμό 144 στη συμβολοσειρά '144', χρησιμοποιείτε τον ενσωματωμένο τύπο κατασκευής `str()`. Εάν θέλετε μια δεκαεξαδική ή οκταδική αναπαράσταση, χρησιμοποιήστε τις ενσωματωμένες συναρτήσεις `hex()` ή `oct()`. Για φανταχτερή μορφοποίηση, βλ. τις ενότητες `f-strings` και `formatstrings` π.χ. `"{:04d}".format(144)` αποδίδει '0144' και `"{: .3f}".format(1.0/3.0)` αποδίδει '0.333'.

### 2.3.6 Πώς μπορώ να τροποποιήσω μια συμβολοσειρά στη θέση της;

Δεν μπορείτε, γιατί οι συμβολοσειρές είναι αμετάβλητες. Στις περισσότερες περιπτώσεις θα πρέπει απλώς να δημιουργήσετε μια νέα συμβολοσειρά από τα διάφορα μέρη από τα οποία θέλετε να τη συναρμολογήσετε. Ωστόσο, εάν χρειάζεστε ένα αντικείμενο με δυνατότητα τροποποίησης δεδομένων `unicode`, δοκιμάστε να χρησιμοποιήσετε ένα αντικείμενο `io.StringIO` ή το module `array`:

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('u', s)
>>> print(a)
array('u', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('u', 'yello, world')
>>> a.tounicode()
'yello, world'
```

### 2.3.7 Πως μπορώ να χρησιμοποιήσω συμβολοσειρές για να καλέσω συναρτήσεις/μεθόδους;

Υπάρχουν διάφορες τεχνικές.

- Το καλύτερο είναι να χρησιμοποιήσετε ένα λεξικό που αντιστοιχίζει συμβολοσειρές σε συναρτήσεις. Το κύριο πλεονέκτημα αυτής της τεχνικής είναι ότι οι συμβολοσειρές δεν χρειάζεται να ταιριάζουν με τα ονόματα των συναρτήσεων. Αυτή είναι επίσης η κύρια τεχνική που χρησιμοποιείται για την εξομοίωση μιας κατασκευής πεζών-κεφαλαίων:

```
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs
```

(συνέχεια στην επόμενη σελίδα)



(συνεχίζεται από την προηγούμενη σελίδα)

```
dispatch[get_input()]() # Note trailing parens to call function
```

- Χρησιμοποιείτε την ενσωματωμένη συνάρτηση `getattr()`:

```
import foo
getattr(foo, 'bar')()
```

Σημειώστε ότι το `getattr()` λειτουργεί σε οποιοδήποτε αντικείμενο, συμπεριλαμβανομένων κλάσεων, οντοτήτων κλάσεων, modules, και ούτω καθεξής.

Αυτό χρησιμοποιείται σε πολλά σημεία της τυπικής βιβλιοθήκης, όπως αυτό:

```
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

- Χρησιμοποιήστε το `locals()` για να επιλύσετε το όνομα της συνάρτησης:

```
def myFunc():
    print("hello")

fname = "myFunc"

f = locals()[fname]
f()
```

### 2.3.8 Is there an equivalent to Perl's `chomp()` for removing trailing newlines from strings?

Μπορείτε να χρησιμοποιήσετε το `S.rstrip("\r\n")` για να αφαιρέσετε όλες τις εμφανίσεις οποιουδήποτε terminator γραμμής από το τέλος της συμβολοσειράς `S` χωρίς να αφαιρέσετε άλλα κενά. Η συμβολοσειρά `S` αντιπροσωπεύει περισσότερες από μία γραμμές στο τέλος, οι terminators γραμμής για όλες τις κενές γραμμές θα αφαιρεθούν:

```
>>> lines = ("line 1 \r\n"
...          "\r\n"
...          "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

Δεδομένου ότι αυτό είναι συνήθως επιθυμητό μόνο κατά την ανάγνωση κειμένου μία γραμμή τη φορά, η χρήση του `S.rstrip()` λειτουργεί καλά.

### 2.3.9 Is there a `scanf()` or `sscanf()` equivalent?

Όχι ως τέτοιο.

For simple input parsing, the easiest approach is usually to split the line into whitespace-delimited words using the `split()` method of string objects and then convert decimal strings to numeric values using `int()` or `float()`. `split()` supports an optional «sep» parameter which is useful if the line uses something other than whitespace as a separator.

Για πιο περίπλοκη ανάλυση εισόδου, τα regular expressions είναι πιο ισχυρά από τις `sscanf` της C και είναι πιο κατάλληλες για την εργασία.

### 2.3.10 What does “UnicodeDecodeError” or “UnicodeEncodeError” error mean?

Βλ το `unicode-howto`.

### 2.3.11 Μπορώ να τερματίσω μια ακατέργαστη συμβολοσειρά με περιττό αριθμό backslashes;

Μια ακατέργαστη συμβολοσειρά που τελειώνει με περιττό αριθμό backslashes θα ξεφύγει από το απόσπασμα της συμβολοσειράς:

```
>>> r'C:\this\will\not\work\'
File "<stdin>", line 1
    r'C:\this\will\not\work\'
      ^
SyntaxError: unterminated string literal (detected at line 1)
```

Υπάρχουν αρκετές λύσεις για αυτό. Ο ένας είναι να χρησιμοποιήσετε κανονικές συμβολοσειρές και να διπλασιάσετε τα backslashes:

```
>>> 'C:\\this\\will\\work\\'
'C:\\this\\will\\work\\'
```

Ένα άλλο είναι να συνδέσετε μια κανονική συμβολοσειρά που περιέχει ένα escaped backslash στην ακατέργαστη συμβολοσειρά:

```
>>> r'C:\this\will\work' '\\'
'C:\\this\\will\\work\\'
```

Είναι επίσης δυνατό να χρησιμοποιήσετε το `os.path.join()` για να προσθέσετε ένα backslash στα Windows:

```
>>> os.path.join(r'C:\this\will\work', '')
'C:\\this\\will\\work\\'
```

Λάβετε υπόψη ότι ενώ ένα backslash θα «escape» από ένα εισαγωγικό για τον προσδιορισμό του σημείου που τελειώνει η ακατέργαστη συμβολοσειρά, δεν υπάρχει διαφυγή κατά την ερμηνεία της τιμής της ακατέργαστης συμβολοσειράς. Δηλαδή, το backslash παραμένει παρόν στην τιμή της ακατέργαστης συμβολοσειράς:

```
>>> r'backslash\'preserved'
"backslash\\'preserved"
```

Δείτε επίσης την προδιαγραφή στην `language reference`.

## 2.4 Απόδοση

### 2.4.1 Το πρόγραμμά μου είναι πολύ αργό. Πως μπορώ να το επιταχύνω;

Αυτό είναι δύσκολο, γενικά. Πρώτον, εδώ είναι μια λίστα με πράγματα που πρέπει να θυμάστε πριν βουτήξετε περαιτέρω:

- Τα χαρακτηριστικά απόδοσης διαφέρουν μεταξύ των υλοποιήσεων Python. Αυτή η FAQ εστιάζει στο *CPython*.
- Η συμπεριφορά μπορεί να διαφέρει μεταξύ των λειτουργικών συστημάτων, ειδικά όταν μιλάμε για I/O ή multi-threading.
- Θα πρέπει πάντα να βρίσκετε τα hot spot στο πρόγραμμά σας πριν επιχειρήσετε να βελτιστοποιήσετε οποιονδήποτε κώδικα (βλ. το module `profile`).
- Η σύνταξη σεναρίων συγκριτικής αξιολόγησης θα σας επιτρέψει να κάνετε iterate γρήγορα κατά την αναζήτηση βελτιώσεων (βλ. το module `timeit`).
- Συνίσταται ανεπιφύλακτα να έχετε καλή κάλυψη κώδικα (μέσω unit testing ή οποιασδήποτε άλλης τεχνικής) πριν από την πιθανή εισαγωγή κρυμμένων παλινδρομήσεων (regressions) σε εξελιγμένες βελτιστοποιήσεις.

Τούτου λεχθέντος, υπάρχουν πολλά κόλπα για την επιτάχυνση του κώδικα Python. Ακολουθούν ορισμένες γενικές αρχές που βοηθούν πολύ στην επίτευξη αποδεκτών επιπέδων απόδοσης:

- Το να κάνετε τους αλγορίθμους σας πιο γρήγορους (ή να αλλάξετε σε ταχύτερους) μπορεί να αποφέρει πολύ μεγαλύτερα οφέλη από το να προσπαθείτε να σκορπίσετε κόλπα μικρό βελτιστοποίησης σε όλο τον κώδικά σας.
- Χρησιμοποιήστε τις σωστές δομές δεδομένων. Μελετήστε την τεκμηρίωση για το builtin-types και το module `collections`.
- Όταν η τυπική βιβλιοθήκη παρέχει ένα πρωτόγονο για να κάνετε κάτι, είναι πιθανό (αν και δεν είναι εγγυημένο) να είναι πιο γρήγορο από οποιαδήποτε εναλλακτική λύση που μπορείτε να βρείτε. Αυτό ισχύει διπλά για πρωτόγονα γραμμένα σε C, όπως ενσωματωμένα και ορισμένους τύπους επεκτάσεων. Για παράδειγμα, φροντίστε να χρησιμοποιήσετε είτε την ενσωματωμένη μέθοδο `list.sort()` είτε τη σχετική συνάρτηση `sorted()` για να κάνετε ταξινομήση (και δείτε το `sortinghowto` για παραδείγματα μέτρησης προηγμένης χρήσης).
- Οι αφαιρέσεις τείνουν να δημιουργούν έμμεσες κατευθύνσεις και αναγκάζουν τον διερμηνέα να εργαστεί περισσότερο. Εάν τα επίπεδα της έμμεσης κατεύθυνσης υπερτερούν του όγκου της χρήσιμης εργασίας που γίνεται, το πρόγραμμά σας θα είναι πιο αργό. Θα πρέπει να αποφύγετε την υπερβολική αφαίρεση, ειδικά με τη μορφή μικροσκοπικών συναρτήσεων ή μεθόδων (που είναι επίσης συχνά επιζήμια για την αναγνωσιμότητα).

Εάν έχετε φτάσει στο όριο του τι μπορεί να επιτρέψει η καθαρή Python, υπάρχουν εργαλεία που θα σας απομακρύνουν. Για παράδειγμα, το *Cython* μπορεί να μεταγλωττίσει μια ελαφρώς τροποποιημένη έκδοση του κώδικα Python σε μια επέκταση C, και μπορεί να χρησιμοποιηθεί σε πολλές διαφορετικές πλατφόρμες. Η Cython μπορεί να εκμεταλλευτεί την μεταγλώττιση (και τους προαιρετικούς σχολιασμούς) για να κάνει τον κώδικά σας πολύ πιο γρήγορο από όταν ερμηνεύεται. Εάν είστε σίγουροι για τις δεξιότητές σας στον προγραμματισμό C, μπορείτε επίσης να write a C extension module μόνοι σας.

**Δείτε επίσης:**

Η σελίδα wiki που είναι αφιερωμένη σε *συμβουλές απόδοσης*.

## 2.4.2 Ποιος είναι ο πιο αποτελεσματικός τρόπος για να συνδέσετε πολλές συμβολοσειρές μεταξύ τους;

Τα αντικείμενα `str` και `bytes` είναι αμετάβλητα, επομένως η σύνδεση πολλών συμβολοσειρών μεταξύ τους είναι αναποτελεσματική καθώς κάθε συνένωση δημιουργεί ένα νέο αντικείμενο. Στη γενική περίπτωση, το συνολικό κόστος χρόνου εκτέλεσης είναι τετραγωνικό στο συνολικό μήκος συμβολοσειράς.

Για να συγκεντρώσετε πολλά αντικείμενα `str`, το προτεινόμενο ιδίωμα είναι να τα τοποθετήσετε σε μια λίστα και να καλέσετε το `str.join()` τέλος:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

(ένα άλλο λογικά αποτελεσματικό ιδίωμα είναι να χρησιμοποιήσετε το `io.StringIO`)

Για τη συγκέντρωση πολλών αντικειμένων `bytes`, το συνιστώμενο ιδίωμα είναι η επέκταση ενός αντικειμένου `bytearray` χρησιμοποιώντας επιτόπια συνένωση (ο τελεστής `+=`):

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

## 2.5 Ακολουθίες (Πλειάδες/Λίστες)

### 2.5.1 Πως μπορώ να κάνω μετατροπή μεταξύ πλειάδων και λιστών;

Ο κατασκευαστής τύπου `tuple(seq)` μετατρέπει οποιαδήποτε ακολουθία (στην πραγματικότητα οποιοδήποτε `iterable`) σε πλειάδα με τα ίδια στοιχεία στην ίδια σειρά.

Για παράδειγμα, το `tuple([1, 2, 3])` αποδίδει `(1, 2, 3)` και το `tuple('abc')` αποδίδει `('a', 'b', 'c')`. Εάν το όρισμα είναι πλειάδα, δεν δημιουργεί αντίγραφο αλλά επιστρέφει το ίδιο αντικείμενο, επομένως είναι φτηνό να καλέσετε το `tuple()` όταν δεν είστε σίγουροι ότι ένα αντικείμενο είναι ήδη πλειάδα.

Ο κατασκευαστής τύπων `list(seq)` μετατρέπει οποιαδήποτε ακολουθία ή `iterable` σε μια λίστα με τα ίδια στοιχεία στην ίδια σειρά. Για παράδειγμα, το `list([1, 2, 3])` αποδίδει `[1, 2, 3]` και `list('abc')` αποδίδει `['a', 'b', 'c']`. Αν το όρισμα είναι λίστα, δημιουργεί απλώς ένα αντίγραφο όπως θα έκανε το `seq[:]`.

### 2.5.2 Τι είναι αρνητικός δείκτης;

Οι ακολουθίες Python `indexed` με θετικούς αριθμούς και αρνητικούς αριθμούς. Για θετικούς αριθμούς το 0 είναι ο πρώτος δείκτης 1 είναι ο δεύτερος δείκτης και ούτω καθεξής. Για αρνητικούς δείκτες το -1 είναι ο τελευταίος δείκτης και το -2 είναι ο προτελευταίος (δίπλα στο τελευταίο) δείκτης και ούτω καθεξής. Σκεφτείτε το `seq[-n]` ως το ίδιο με το `seq[len(seq)-n]`.

Η χρήση αρνητικών δεικτών μπορεί να είναι πολύ βολική. Για παράδειγμα `s[:-1]` είναι όλη η συμβολοσειρά εκτός από τον τελευταίο χαρακτήρα της, ο οποίος είναι χρήσιμος για την αφαίρεση της νέας γραμμής που ακολουθεί μια συμβολοσειρά.

### 2.5.3 Πώς μπορώ να επαναλάβω μια ακολουθία με αντίστροφη σειρά;

Χρησιμοποιείτε την ενσωματωμένη συνάρτηση `reversed()`:

```
for x in reversed(sequence):
    ... # do something with x ...
```

Αυτό δεν θα επηρεάσει την αρχική σας ακολουθία, αλλά δημιουργήστε ένα νέο αντίγραφο με αντίστροφη σειρά για επανάληψη.

### 2.5.4 Πως αφαιρείτε διπλότυπα από μια λίστα;

Δείτε το Python Cookbook για μια μακρά συζήτηση σχετικά με πολλούς τρόπους για να το κάνετε αυτό:

<https://code.activestate.com/recipes/52560/>

Εάν δεν σας πειράζει να αναδιατάξετε τη λίστα, ταξινομήστε την και μετά σαρώστε από το τέλος της λίστας, διαγράφοντας τα διπλότυπα καθώς προχωράτε:

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

Εάν όλα τα στοιχεία της λίστας μπορούν να χρησιμοποιηθούν ως κλειδιά συνόλου (δηλαδή είναι όλα τα *hashable*) αυτό είναι συχνά πιο γρήγορο:

```
mylist = list(set(mylist))
```

Αυτό μετατρέπει τη λίστα σε ένα σύνολο, αφαιρώντας έτσι τα διπλότυπα και στη συνέχεια ξανά σε λίστα.

### 2.5.5 Πως αφαιρείτε πολλαπλά στοιχεία από μία λίστα

Όπως και με την κατάργηση των διπλότυπων, το ρητό `iterating` αντίστροφα με μια συνθήκη διαγραφής είναι μια πιθανότητα. Ωστόσο, είναι ευκολότερο και πιο γρήγορο να χρησιμοποιήσετε την αντικατάσταση τμημάτων με ένα έμμεσο ή ρητώς προς τα εμπρός `iteration`. Ακολουθούν τρεις παραλλαγές:

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

Το `comprehension` της λίστας μπορεί να είναι ταχύτερο.

## 2.5.6 Πως μπορείτε να φτιάξετε έναν πίνακα στην Python;

Χρησιμοποιήστε μια λίστα:

```
[ "this", 1, "is", "an", "array" ]
```

Οι λίστες είναι ισοδύναμες με τους πίνακες της C ή Pascal στην χρονική τους πολυπλοκότητα• η κύρια διαφορά είναι ότι μια λίστα Python μπορεί να περιέχει αντικείμενα πολλών διαφορετικών τύπων.

Το module `array` παρέχει επίσης μεθόδους για τη δημιουργία πινάκων σταθερών τύπων με συμπαγείς αναπαραστάσεις, αλλά είναι πιο αργές στην ευρετηρίαση από τις λίστες. Σημειώστε επίσης ότι το `NumPy` και άλλο πακέτα τρίτων, ορίζουν δομές τύπους `array` με διάφορα χαρακτηριστικά επίσης.

Για να λάβετε συνδεδεμένες λίστες τύπου Lisp, μπορείτε να εξομοιώσετε *cons* κελιά χρησιμοποιώντας πλειάδες:

```
lisp_list = ("like", ("this", ("example", None) ) )
```

Εάν είναι επιθυμητή η μεταβλητότητα, μπορείτε να χρησιμοποιήσετε λίστες αντί για πλειάδες. Εδώ το ανάλογο ενός Lisp *car* είναι `lisp_list[0]` και το ανάλογο του *cdr* είναι `lisp_list[1]`. Μόνο κάντε το αν είστε βέβαιοι ότι πραγματικά χρειάζεται, γιατί είναι συνήθως πιο αργό και από τη χρήση λιστών Python.

## 2.5.7 Πως φτιάχνω μια πολυδιάστατη λίστα;

Μάλλον προσπαθήσατε να φτιάξετε έναν πολυδιάστατο πίνακα σαν αυτόν:

```
>>> A = [ [None] * 2 ] * 3
```

Αυτό φαίνεται σωστό αν το εκτυπώσετε:

```
>>> A
[[None, None], [None, None], [None, None]]
```

Αλλά όταν εκχωρείτε μια τιμή, εμφανίζεται σε πολλά σημεία:

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

Ο λόγος είναι ότι η αναπαραγωγή μιας λίστα με `*` δεν δημιουργεί αντίγραφα, δημιουργεί μόνο αναφορές στα υπάρχοντα αντικείμενα. Το `*3` δημιουργεί μια λίστα που περιέχει 3 αναφορές στην ίδια λίστα μήκους δύο. Οι αλλαγές σε μία σειρά θα εμφανίζονται σε όλες τις σειρές, κάτι που σχεδόν σίγουρα δεν είναι αυτό που θέλετε.

Η προτεινόμενη προσέγγιση είναι να δημιουργήσετε πρώτα μια λίστα με το επιθυμητό μήκος και στη συνέχεια να συμπληρώσετε κάθε στοιχείο με μια νέα λίστα:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

Αυτό δημιουργεί μια λίστα που περιέχει 3 διαφορετικές λίστες με μήκος δύο. Μπορείτε επίσης να χρησιμοποιήσετε ένα `comprehension` λίστας:

```
w, h = 2, 3
A = [ [None] * w for i in range(h) ]
```

Εναλλακτικά, μπορείτε να χρησιμοποιήσετε μια επέκταση που παρέχει έναν τύπο δεδομένων μήτρας (*matrix*)• Το `NumPy` είναι το πιο γνωστό.

## 2.5.8 Πώς μπορώ να εφαρμόσω μια μέθοδο ή μια συνάρτηση σε μια ακολουθία αντικειμένων;

Για να καλέσετε μια μέθοδο ή μια συνάρτηση και να συγκεντρώσετε τις επιστρεφόμενες τιμές είναι μια λίστα, ένα *list comprehension* είναι μια κομψή λύση:

```
result = [obj.method() for obj in mylist]

result = [function(obj) for obj in mylist]
```

Για να εκτελέσετε απλώς τη μέθοδο ή τη συνάρτηση χωρίς να αποθηκεύσετε τις επιστρεφόμενες τιμές, αρκεί ένας απλός βρόχος `for`:

```
for obj in mylist:
    obj.method()

for obj in mylist:
    function(obj)
```

## 2.5.9 Γιατί το `a_tuple[i] += ["item"]` δημιουργεί μια εξαίρεση όταν λειτουργεί η προσθήκη;

Αυτό οφείλεται σε έναν συνδυασμό του γεγονότος ότι οι επαυξημένοι τελεστές εκχώρησης είναι τελεστές *εκχώρησης* και της διαφοράς μεταξύ μεταβλητών και αμετάβλητων αντικειμένων στην Python.

Αυτή η συζήτηση ισχύει γενικά όταν οι επαυξημένοι τελεστές εκχώρησης εφαρμόζονται σε στοιχεία μιας πλειάδας που δείχνουν σε μεταβλητά αντικείμενα, αλλά θα χρησιμοποιήσουμε `list` και `+=` ως υπόδειγμά μας.

Εάν γράψετε:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

Ο λόγος για την εξαίρεση θα πρέπει να είναι αμέσως σαφής: το 1 προστίθεται στο αντικείμενο `a_tuple[0]` δείχνει στο (1), παράγοντας το αντικείμενο αποτέλεσμα, 2, αλλά όταν προσπαθούμε να αντιστοιχίσουμε το αποτέλεσμα του υπολογισμού, 2, στο στοιχείο 0 της πλειάδας, λαμβάνουμε ένα σφάλμα επειδή δεν μπορούμε να αλλάξουμε αυτό που δείχνει ένα στοιχείο μιας πλειάδας.

Κάτω από τα καλύμματα, αυτό που κάνει αυτή η επαυξημένη δήλωση ανάθεσης είναι περίπου το εξής:

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

Είναι το τμήμα ανάθεσης της λειτουργίας που παράγει το σφάλμα, αφού μια πλειάδα είναι αμετάβλητη.

Όταν γράφετε κάτι σαν:

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
...
TypeError: 'tuple' object does not support item assignment
```

Η εξαίρεση είναι λίγο πιο εκπληκτική, και ακόμη πιο εκπληκτικό είναι το γεγονός ότι παρόλο που υπήρχε ένα σφάλμα, το παράρτημα λειτούργησε:

```
>>> a_tuple[0]
['foo', 'item']
```

Για να δείτε γιατί συμβαίνει αυτό, πρέπει να γνωρίζετε ότι (α) εάν ένα αντικείμενο υλοποιεί μια μαγική μέθοδο `__iadd__()`, που καλείται όταν εκτελείται η επαυξημένη ανάθεση `+=` και η τιμή επιστροφής είναι αυτή που χρησιμοποιείται στη δήλωση εκχώρησης• και (β) για λίστες, `__iadd__()` ισοδυναμεί με την κλήση του `extend()` στη λίστα και επιστρέφει τη λίστα. Για αυτό λέμε ότι για λίστες `+=` είναι μια «συντομογραφία» για `list.extend()`:

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

Αυτό ισοδυναμεί με:

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

Το αντικείμενο στο οποίο υποδεικνύεται από το `a_list` έχει μεταλλαχθεί και ο δείκτης στο μεταλλαγμένο αντικείμενο έχει εκχωρηθεί πίσω στο `a_list`. Το τελικό αποτέλεσμα της ανάθεσης είναι ένα `no-op`, καθώς είναι ένας δείκτης στο ίδιο αντικείμενο που το `a_list` έδειχνε προηγουμένως, αλλά η ανάθεση εξακολουθεί να γίνεται.

Έτσι, στο παράδειγμά μας, αυτό που συμβαίνει είναι ισοδύναμο με:

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

Το `__iadd__()` πετυχαίνει, και έτσι η λίστα επεκτείνεται, αλλά παρόλο που το αποτέλεσμα δείχνει στο ίδιο αντικείμενο που δείχνει ήδη το `a_tuple[0]`, αυτή η τελική ανάθεση εξακολουθεί να έχει ως αποτέλεσμα ένα λάθος, γιατί οι πλειάδες είναι αμετάβλητες.

## 2.5.10 Θέλω να κάνω μια περίπλοκη ταξινόμηση: μπορείτε να κάνετε ένα Schwartzian Transform στην Python;

Η τεχνική, που αποδίδεται στον Randal Schwartz της κοινότητας Perl, ταξινομεί τα στοιχεία μιας λίστας με βάση μια μέτρηση που αντιστοιχίζει κάθε στοιχείο στην «τιμή ταξινόμησης» του. Στην Python, χρησιμοποιήστε το όρισμα `key` για τη μέθοδο `list.sort()`:

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```



### 2.5.11 Πως μπορώ να ταξινομήσω μια λίστα με βάση τις τιμές από μια άλλη λίστα;

Συγχωνεύστε τα σε έναν iterator πλειάδων, ταξινομήστε τη λίστα που προκύπτει και, στην συνέχεια επιλέξτε το στοιχείο που θέλετε.

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[('I'm', 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

## 2.6 Αντικείμενα

### 2.6.1 Τι είναι μια κλάση;

Μια κλάση είναι ο συγκεκριμένος τύπος αντικειμένου που δημιουργείται με την εκτέλεση μιας δήλωσης κλάσης. Τα αντικείμενα κλάσης χρησιμοποιούνται ως πρότυπα για τη δημιουργία αντικειμένων παρουσίας, τα οποία ενσωματώνουν τόσο τα δεδομένα (χαρακτηριστικά) όσο και τον κώδικα (μεθόδους) ειδικά για έναν τύπο δεδομένων.

Μια κλάση μπορεί να βασίζεται σε μία ή περισσότερες άλλες κλάσεις, που ονομάζονται βασικές κλάσεις της. Στη συνέχεια κληρονομεί τα χαρακτηριστικά και τις μεθόδους των βασικών κλάσεων. Αυτό επιτρέπει σε ένα μοντέλο αντικειμένου να βελτιωθεί διαδοχικά με κληρονομικότητα. Μπορεί να έχετε μια γενική κλάση Mailbox που παρέχει βασικές μεθόδους πρόσβασης για ένα γραμματοκιβώτιο και υποκλάσεις όπως MboxMailbox, MaildirMailbox, OutlookMailbox που χειρίζονται διάφορες συγκεκριμένες μορφές γραμματοκιβωτίου.

### 2.6.2 Τι είναι μια μέθοδος;

Μια μέθοδος είναι μια συνάρτηση σε κάποιο αντικείμενο `x` που συνήθως καλείτε ως `x.name (ορίσματα...)`. Οι μέθοδοι ορίζονται ως συναρτήσεις εντός του ορισμού κλάσης:

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

### 2.6.3 Τι είναι το self;

Το `self` είναι απλώς ένα συμβατικό όνομα για το πρώτο όρισμα μιας μεθόδου. Μια μέθοδος που ορίζεται ως `meth(self, a, b, c)` πρέπει να ονομάζεται `x.meth(a, b, c)` για κάποιο παράδειγμα `x` της κλάσης στην οποία εμφανίζεται ο ορισμός· η καλούμενη μέθοδος θα ονομάζεται `meth(x, a, b, c)`.

Βλ. επίσης *Why must "self" be used explicitly in method definitions and calls?*.

## 2.6.4 Πώς μπορώ να ελέγξω εάν ένα αντικείμενο είναι μια οντότητα μιας δεδομένης κλάσης ή μιας υποκλάσης της;

Χρησιμοποιήστε την ενσωματωμένη συνάρτηση `isinstance(obj, cls)`. Μπορείτε να ελέγξετε εάν ένα αντικείμενο είναι μια παρουσία οποιασδήποτε από έναν αριθμό κλάσεων παρέχοντας μια πλειάδα αντί για μια μεμονωμένη κλάση, π.χ. `isinstance(obj, (class1, class2, ...))`, και μπορεί επίσης να ελέγξει εάν ένα αντικείμενο είναι ένας από τους ενσωματωμένους τύπους της Python, π.χ. `isinstance(obj, str)` ή `isinstance(obj, (int, float, complex))`.

Λάβετε υπόψη ότι το `isinstance()` ελέγχει επίσης για εικονική κληρονομικότητα από μια *abstract base class*. Έτσι, η δοκιμή θα επιστρέψει `True` για μια εγγεγραμμένη κλάση ακόμα κι αν δεν έχει κληρονομήσει άμεσα ή έμμεσα από αυτό. Για να ελέγξετε μια «αληθινή κληρονομικότητα», σαρώστε το *MRO* της κλάσης:

```
from collections.abc import Mapping
```

```
class P:
    pass
```

```
class C(P):
    pass
```

```
Mapping.register(P)
```

```
>>> c = C()
>>> isinstance(c, C)           # direct
True
>>> isinstance(c, P)           # indirect
True
>>> isinstance(c, Mapping)     # virtual
True

# Actual inheritance chain
>>> type(c).__mro__
(<class 'C'>, <class 'P'>, <class 'object'>)

# Test for "true inheritance"
>>> Mapping in type(c).__mro__
False
```

Λάβετε υπόψη ότι τα περισσότερα προγράμματα δεν χρησιμοποιούν το `isinstance()` σε κλάσεις που ορίζονται από τη χρήση πολύ συχνά. Εάν αναπτύσσετε μόνοι σας τις κλάσεις, ένα πιο σωστό αντικειμενοστρεφές στυλ είναι να ορίζετε μεθόδους στις κλάσεις που ενσωματώνουν μια συγκεκριμένη συμπεριφορά, αντί να ελέγχετε την κλάση του αντικειμένου και να κάνετε κάτι διαφορετικό με βάση την κλάση που είναι, για παράδειγμα, εάν έχετε μια συνάρτηση που κάνει κάτι:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ... # code to search a mailbox
    elif isinstance(obj, Document):
        ... # code to search a document
    elif ...
```

Μια καλύτερη προσέγγιση είναι να ορίσετε μια μέθοδο `search()` σε όλες τις κλάσεις και απλώς να την καλέσετε:

```
class Mailbox:
    def search(self):
        ... # code to search a mailbox

class Document:
    def search(self):
        ... # code to search a document

obj.search()
```

## 2.6.5 Τι είναι το delegation;

Το delegation είναι μια αντικειμενοστραφής τεχνική (ονομάζεται επίσης μοτίβο σχεδίασης). Ας υποθέσουμε ότι έχετε ένα αντικείμενο `x` και θέλετε να αλλάξετε τη συμπεριφορά μιας μόνο από τις μεθόδους του. Μπορείτε να δημιουργήσετε μια νέα κλάση που παρέχει μια νέα υλοποίηση της μεθόδου που σας ενδιαφέρει να αλλάξετε και εκχωρεί όλες τις άλλες μεθόδους στην αντίστοιχη μέθοδο του `x`.

Οι προγραμματιστές Python μπορούν εύκολα να υλοποιήσουν την ανάθεση. Για παράδειγμα, η ακόλουθη κλάση υλοποιεί μια κλάση που συμπεριφέρεται σαν αρχείο αλλά μετατρέπει όλα τα γραπτά δεδομένα σε κεφαλαία:

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

Εδώ η κλάση `UpperOut` επαναπροσδιορίζει τη μέθοδο `write()` για να μετατρέψει τη συμβολοσειρά ορίσματος σε κεφαλαία πριν καλέσει την υποκείμενη μέθοδο `self._outfile.write()`. Όλες οι άλλες μέθοδοι εκχωρούνται στο υποκείμενο αντικείμενο `self._outfile`. Το delegation ολοκληρώνεται μέσω της μεθόδου `__getattr__()`. Συμβουλευτείτε το [the language reference](#) για περισσότερες πληροφορίες σχετικά με τον έλεγχο της πρόσβασης.

Λάβετε υπόψη ότι για πιο γενικές περιπτώσεις η ανάθεση μπορεί να γίνει πιο δύσκολη. Όταν τα χαρακτηριστικά πρέπει να οριστούν καθώς και να ανακτηθούν, η κλάση πρέπει να ορίσει μια μέθοδο `__setattr__()` επίσης, και πρέπει να το κάνει προσεκτικά. Η βασική υλοποίηση του `__setattr__()` είναι περίπου ισοδύναμο με το εξής:

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

Most `__setattr__()` implementations must modify `self.__dict__` to store local state for self without causing an infinite recursion.

## 2.6.6 Πώς μπορώ να καλέσω μια μέθοδο που ορίζεται σε μια βασική κλάση από μια παράγωγη κλάση που την επεκτείνει;

Χρησιμοποιήστε την ενσωματωμένη συνάρτηση `super()`:

```
class Derived(Base):
    def meth(self):
        super().meth() # calls Base.meth
```

Στο παράδειγμα, το `super()` θα προσδιορίσει αυτόματα το στιγμιότυπο από το οποίο κλήθηκε (η τιμή `self`), αναζητήστε τη *method resolution order* (MRO) με `type(self).__mro__`, και επιστρέψτε το επόμενο στη σειρά μετά το `Derived` στο MRO: `Base`.

## 2.6.7 Πως μπορώ να οργανώσω τον κώδικα μου προκειμένου να διευκολύνω την αλλαγή της βασικής κλάσης;

Θα μπορούσατε να αντιστοιχίσετε τη βασική κλάση σε ένα ψευδώνυμο και να προκύψει το ψευδώνυμο. Στην συνέχεια, το μόνο που πρέπει να αλλάξετε είναι η τιμή που έχει εκχωρηθεί ψευδώνυμο. Παρεμπιπτόντως, αυτό το κόλπο είναι επίσης χρήσιμο εάν θέλετε να αποφασίσετε δυναμικά (π.χ. ανάλογα με την διαθεσιμότητα των πόρων) ποια βασική κλάση να χρησιμοποιήσετε Παράδειγμα:

```
class Base:
    ...

BaseAlias = Base

class Derived(BaseAlias):
    ...
```

## 2.6.8 Πως δημιουργώ δεδομένα στατικής κλάσης και μεθόδους στατικής κλάσης;

Τόσο τα στατιστικά δεδομένα όσο και οι στατικές μέθοδοι (με την έννοια της C++ ή της Java) υποστηρίζονται στην Python.

Για στατικά δεδομένα, απλώς ορίστε ένα χαρακτηριστικό κλάσης. Για να εκχωρήσετε μια νέα τιμή στο χαρακτηριστικό, πρέπει να χρησιμοποιήσετε ρητά το όνομα κλάσης στην εκχώρηση:

```
class C:
    count = 0 # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count # or return self.count
```

Το `c.count` αναφέρεται επίσης στο `C.count` για οποιοδήποτε `c`, έτσι ώστε να ισχύει το `isinstance(c, C)`, εκτός εάν παρακαμφθεί από το ίδιο το `c` ή από κάποια κλάση στη διαδρομή αναζήτησης της βασικής κλάσης από το `c.__class__` πίσω στο `C`.

Προσοχή: σε μια μέθοδο του `C`, μια ανάθεση όπως `self.count = 42` δημιουργεί μια νέα και άσχετη παρουσία με το όνομα «count» στο δικό του dict του `self`. Επανασύνδεση μιας κλάσης-στατικής όνομα δεδομένων πρέπει πάντα να προσδιορίζει την κλάση είτε βρίσκεται μέσα σε μια μέθοδο είτε όχι:

```
C.count = 314
```

Οι στατικές μέθοδοι είναι δυνατές:

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

Ωστόσο, ένας πολύ πιο απλός τρόπος για να λάβετε το αποτέλεσμα μιας στατικής μεθόδου είναι μέσω μιας απλής συνάρτησης σε επίπεδο μονάδας:

```
def getcount():
    return C.count
```

Εάν ο κώδικας σας είναι δομημένος έτσι ώστε να ορίζει μία κλάση (ή στενά συνδεδεμένη ιεραρχίας κλάσεων) ανά module, αυτό παρέχει την επιθυμητή ενθυλάκωση.

## 2.6.9 Πως μπορώ να υπερφορτώσω κατασκευαστές (ή μεθόδους) στην Python;

Αυτή η απάντηση ισχύει στην πραγματικότητα για όλες τις μεθόδους, αλλά η ερώτηση συνήθως εμφανίζεται πρώτη στο πλαίσιο των κατασκευαστών.

Στην C++ θα γράφατε

```
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

Στην Python πρέπει να γράψετε έναν μοναδικό κατασκευαστή που να πιάνει όλες τις περιπτώσεις χρησιμοποιώντας προεπιλεγμένα ορίσματα. Για παράδειγμα:

```
class C:
    def __init__(self, i=None):
        if i is None:
            print("No arguments")
        else:
            print("Argument is", i)
```

Αυτό δεν είναι εντελώς ισοδύναμο, αλλά αρκετά κοντά στην πράξη.

Θα μπορούσατε επίσης να δοκιμάσετε μια λίστα ορισμάτων μεταβλητού μήκους, π.χ.

```
def __init__(self, *args):
    ...
```

Η ίδια προσέγγιση λειτουργεί για όλους τους ορισμούς μεθόδων.

### 2.6.10 Προσπαθώ να χρησιμοποιήσω `__spam` και λαμβάνω ένα σφάλμα σχετικά με το `_SomeClassName__spam`.

Τα ονόματα μεταβλητών με διπλή υπογράμμιση στην αρχή είναι «mangled» για να παρέχουν έναν απλό αλλά αποτελεσματικό τρόπο ορισμού των ιδιωτικών μεταβλητών κλάσης. Οποιοδήποτε αναγνωριστικό της φόρμας `__spam` (τουλάχιστον δύο προπορευόμενες κάτω παύλες, το πολύ ένα τέλος υπογράμμισης) αντικαθίσταται μέσω κειμένου το `_classname__spam`, όπου το `classname` είναι το τρέχον όνομα κλάσης με απογυμνωμένες τυχόν προηγούμενες παύλες.

This doesn't guarantee privacy: an outside user can still deliberately access the «`_classname__spam`» attribute, and private values are visible in the object's `__dict__`. Many Python programmers never bother to use private variable names at all.

### 2.6.11 Η κλάση μου ορίζει `__del__` αλλά δεν καλείται όταν διαγράψω το αντικείμενο.

Υπάρχουν διάφοροι πιθανοί λόγοι για αυτό.

Η πρόταση `del` δεν καλεί απαραίτητως το `__del__()` – απλώς μειώνει τον αριθμό αναφοράς του αντικειμένου, και αν αυτό φτάσει στο μηδέν καλείται το `__del__()`.

Εάν οι δομές δεδομένων σας περιέχουν κυκλικούς συνδέσμους (π.χ. ένα δέντρο όπου κάθε παιδί έχει μια αναφορά γονέα και κάθε γονέας έχει μια λίστα παιδιών), οι μετρήσεις δεν θα επανέλθουν ποτέ στο μηδέν. Κάθε τόσο η Python εκτελεί έναν αλγόριθμο για να ανιχνεύσει τέτοιους κύκλους, αλλά ο συλλέκτης σκουπιδιών μπορεί να εκτελεστεί κάποια στιγμή μετά την εξαφάνιση της τελευταίας αναφοράς στη δομή δεδομένων σας, επομένως η μέθοδος `__del__()` μπορεί να κληθεί σε μια άβολη και τυχαία στιγμή. Αυτό δεν είναι βολικό εάν προσπαθείτε να αναπαράξετε ένα πρόβλημα. Ακόμη χειρότερα, η σειρά με την οποία εκτελούνται μέθοδοι `__del__()` του αντικειμένου είναι αυθαίρετη. Μπορείτε να εκτελέσετε το `gc.collect()` για να αναγκάσετε μια συλλογή, αλλά υπάρχουν παθολογικές περιπτώσεις όπου τα αντικείμενα δεν θα συλλεχθούν ποτέ.

Παρά τον συλλέκτη κύκλου, εξακολουθεί να είναι καλή ιδέα να είναι καλή ιδέα να ορίσετε μια ρητή μέθοδο `close()` σε αντικείμενα που θα καλούνται κάθε φορά που τελειώνετε με αυτά. Η μέθοδος `close()` μπορεί στη συνέχεια να αφαιρεθεί χαρακτηριστικά που αναφέρονται σε υποαντικείμενα. Μην καλείτε το `__del__()` απευθείας – `__del__()` θα πρέπει να καλείτε το `close()` και το `close()` θα πρέπει να βεβαιωθεί ότι μπορεί να κληθεί περισσότερες από μία φορές για το ίδιο αντικείμενο.

Ένα άλλος τρόπος για να αποφύγετε τις κυκλικές αναφορές είναι να χρησιμοποιήσετε το module `weakref`, το οποίο σας επιτρέπει να αυξάνετε τον αριθμό των αναφορών τους. Οι δομές δεδομένων δέντρων, για παράδειγμα, θα πρέπει να χρησιμοποιούν αδύναμες αναφορές για τις αναφορές γονέων και αδελφών (αν τα χρειαστούν!).

Τέλος, εάν η μέθοδος `__del__()` εγείρει μια εξαίρεση, εκτυπώνεται ένα προειδοποιητικό μήνυμα στη διεύθυνση `sys.stderr`.

### 2.6.12 Πως μπορώ να λάβω μια λίστα με όλες τις οντότητες μιας δεδομένης κλάσης;

Η Python δεν παρακολουθεί όλες τις παρουσίες μιας κλάσης (ή ενός ενσωματωμένου τύπου). Μπορείτε να προγραμματίσετε τον κατασκευαστή της κλάσης να παρακολουθεί όλες τις οντότητες διατηρώντας μια λίστα αδύναμων αναφορών σε κάθε παρουσία.

### 2.6.13 Γιατί το αποτέλεσμα του `id()` φαίνεται να μην είναι μοναδικό;

Το ενσωματωμένο `id()` επιστρέφει έναν ακέραιο που είναι εγγυημένο ότι είναι μοναδικός κατά τη διάρκεια ζωής του αντικειμένου. Εφόσον στο CPython, αυτή είναι διεύθυνση μνήμης του αντικειμένου, συμβαίνει συχνά ότι μετά τη διαγραφή ενός αντικειμένου από τη μνήμη, το επόμενο πρόσφατα δημιουργημένο αντικείμενο εκχωρείται στην ίδια θέση στη μνήμη. Αυτό φαίνεται από αυτό το παράδειγμα:

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

Τα δύο αναγνωριστικά ανήκουν σε διαφορετικά ακέραια αντικείμενα που δημιουργούνται πριν και διαγράφονται αμέσως μετά την εκτέλεση της κλήσης `id()`. Για να βεβαιωθείτε ότι τα αντικείμενα των οποίων το αναγνωριστικό θέλετε να εξετάσετε είναι ακόμα ζωντανά, δημιουργήστε μια άλλη αναφορά στο αντικείμενο:

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

### 2.6.14 Πότε μπορώ να βασιστώ σε δοκιμές ταυτότητας με τον τελεστή `is`;

Ο τελεστής `is` ελέγχει την ταυτότητα του αντικειμένου. Η δοκιμή `a is b` ισοδυναμεί με `id(a) == id(b)`.

Η πιο σημαντική ιδιότητα ενός τεστ ταυτότητας είναι ότι ένα αντικείμενο είναι πάντα πανομοιότυπο με τον εαυτό του, το `a is a` επιστρέφει πάντα `True`. Τα τεστ ταυτότητας είναι συνήθως ταχύτερα από τα τεστ ισότητας. Και σε αντίθεση με τα τεστ ισότητας, τα τεστ ταυτότητας είναι εγγυημένα ότι θα επιστρέψουν ένα boolean `True` ή `False`.

Ωστόσο, τα τεστ ταυτότητας μπορούν μόνο να αντικαταστήσουν τα τεστ ισότητας όταν είναι εξασφαλισμένη η ταυτότητα αντικειμένου. Γενικά, υπάρχουν τρεις περιπτώσεις όπου η ταυτότητα είναι εγγυημένη:

- 1) Assignments create new names but do not change object identity. After the assignment `new = old`, it is guaranteed that `new is old`.
- 2) Putting an object in a container that stores object references does not change object identity. After the list assignment `s[0] = x`, it is guaranteed that `s[0] is x`.
- 3) If an object is a singleton, it means that only one instance of that object can exist. After the assignments `a = None` and `b = None`, it is guaranteed that `a is b` because `None` is a singleton.

Στις περισσότερες άλλες περιπτώσεις, τα τεστ ταυτότητας δεν ενδείκνυνται και προτιμώνται τα τεστ ισότητας. Ειδικότερα, τα τεστ ταυτότητας δεν θα πρέπει να χρησιμοποιούνται για τον έλεγχο σταθερών όπως `int` και `str` που δεν είναι εγγυημένα singletons:

```
>>> a = 1000
>>> b = 500
>>> c = b + 500
>>> a is c
False

>>> a = 'Python'
>>> b = 'Py'
>>> c = b + 'thon'
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
>>> a is c
False
```

Ομοίως, τα νέα στιγμιότυπα μεταβλητών κοντέινερ δεν είναι ποτέ πανομοιότυπα:

```
>>> a = []
>>> b = []
>>> a is b
False
```

Στον τυπικό κώδικα βιβλιοθήκης, θα δείτε πολλά κοινά μοτίβα για τη σωστή χρήση των δοκιμών ταυτότητας:

1) As recommended by **PEP 8**, an identity test is the preferred way to check for `None`. This reads like plain English in code and avoids confusion with other objects that may have boolean values that evaluate to false.

2) Detecting optional arguments can be tricky when `None` is a valid input value. In those situations, you can create a singleton sentinel object guaranteed to be distinct from other objects. For example, here is how to implement a method that behaves like `dict.pop()`:

```
_sentinel = object()

def pop(self, key, default=_sentinel):
    if key in self:
        value = self[key]
        del self[key]
        return value
    if default is _sentinel:
        raise KeyError(key)
    return default
```

3) Container implementations sometimes need to augment equality tests with identity tests. This prevents the code from being confused by objects such as `float('NaN')` that are not equal to themselves.

Για παράδειγμα, εδώ είναι η υλοποίηση του `collections.abc.Sequence.__contains__()`:

```
def __contains__(self, value):
    for v in self:
        if v is value or v == value:
            return True
    return False
```

## 2.6.15 Πώς μπορεί μια υποκλάση να ελέγξει ποια δεδομένα αποθηκεύονται σε μια αμετάβλητη παρουσία;

Κατά την υποκλάση ενός αμετάβλητου τύπου, παρακάμψτε τη μέθοδο `__new__()` αντί για τη μέθοδο `__init__()`. Η τελευταία εκτελείται μόνο αφού δημιουργηθεί μια παρουσία, η οποία είναι πολύ αργά για να αλλάξει δεδομένα σε μια αμετάβλητη περίπτωση.

Όλες αυτές οι αμετάβλητες κλάσεις έχουν διαφορετική υπογραφή από τη μητρική τους κλάση:

```
from datetime import date

class FirstOfMonthDate(date):
    "Always choose the first day of the month"
    def __new__(cls, year, month, day):
```

(συνέχεια στην επόμενη σελίδα)



(συνεχίζεται από την προηγούμενη σελίδα)

```

        return super().__new__(cls, year, month, 1)

class NamedInt(int):
    "Allow text names for some numbers"
    xlat = {'zero': 0, 'one': 1, 'ten': 10}
    def __new__(cls, value):
        value = cls.xlat.get(value, value)
        return super().__new__(cls, value)

class TitleStr(str):
    "Convert str to name suitable for a URL path"
    def __new__(cls, s):
        s = s.lower().replace(' ', '-')
        s = ''.join([c for c in s if c.isalnum() or c == '-'])
        return super().__new__(cls, s)

```

Οι κλάσεις μπορούν να χρησιμοποιηθούν έτσι:

```

>>> FirstOfMonthDate(2012, 2, 14)
FirstOfMonthDate(2012, 2, 1)
>>> NamedInt('ten')
10
>>> NamedInt(20)
20
>>> TitleStr('Blog: Why Python Rocks')
'blog-why-python-rocks'

```

## 2.6.16 Πώς μπορώ να αποθηκεύσω τις κλήσεις μεθόδου στην κρυφή μνήμη;

Τα δύο βασικά εργαλεία για τις μεθόδους αποθήκευσης στην προσωρινή μνήμη είναι τα `functools.cached_property()` και `functools.lru_cache()`. Το πρώτο αποθηκεύει τα αποτελέσματα σε επίπεδο παρουσίας και το δεύτερο σε επίπεδο κλάσης.

Η προσέγγιση `cached_property` λειτουργεί μόνο με μεθόδους που δεν λαμβάνουν ορίσματα. Δεν δημιουργεί αναφορά στο στιγμιότυπο. Το αποτέλεσμα της `cached` μεθόδους θα διατηρηθεί μόνο όσο το στιγμιότυπο είναι ζωντανό.

Το πλεονέκτημα είναι ότι όταν ένα στιγμιότυπο δεν χρησιμοποιείται πλέον, το αποτέλεσμα της αποθηκευμένης μεθόδου θα απελευθερωθεί αμέσως. Το μειονέκτημα είναι ότι εάν συσσωρευτούν στιγμιότυπα, θα είναι και τα αποτελέσματα της συσσωρευμένης μεθόδου. Μπορούν να αναπτυχθούν χωρίς περιορισμούς.

Η προσέγγιση `lru_cache` λειτουργεί με μεθόδους που έχουν ορίσματα *hashable*. Δημιουργεί μια αναφορά στο στιγμιότυπο, εκτός εάν καταβληθούν ειδικές προσπάθειες για να περάσει σε αδύναμες αναφορές.

Το πλεονέκτημα του αλγορίθμου που χρησιμοποιήθηκε λιγότερο πρόσφατα είναι ότι η κρυφή μνήμη οριοθετείται από το καθορισμένο `maxsize`. Το μειονέκτημα είναι ότι τα στιγμιότυπα διατηρούνται ζωντανά έως ότου παλαιώσουν από την κρυφή μνήμη ή μέχρι να διαγραφεί η κρυφή μνήμη.

Αυτό το παράδειγμα δείχνει τις διάφορες τεχνικές:

```

class Weather:
    "Lookup weather information on a government website"

    def __init__(self, station_id):
        self._station_id = station_id
        # The _station_id is private and immutable

```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```

def current_temperature(self):
    "Latest hourly observation"
    # Do not cache this because old results
    # can be out of date.

@cached_property
def location(self):
    "Return the longitude/latitude coordinates of the station"
    # Result only depends on the station_id

@lru_cache(maxsize=20)
def historic_rainfall(self, date, units='mm'):
    "Rainfall on a given date"
    # Depends on the station_id, date, and units.

```

Το παραπάνω παράδειγμα προϋποθέτει ότι το `station_id` δεν αλλάζει ποτέ. Εάν τα σχετικά χαρακτηριστικά παρουσίας είναι μεταβλητά, η προσέγγιση `cached_property` δεν μπορεί να λειτουργήσει επειδή δεν μπορεί να εντοπίσει αλλαγές στα χαρακτηριστικά.

Για να λειτουργήσει η προσέγγιση `lru_cache` όταν το `station_id` είναι μεταβλητό, η κλάση πρέπει να ορίσει τις μεθόδους `__eq__()` και `__hash__()` ώστε η κρυφή μνήμη να μπορεί να εντοπίσει σχετικές ενημερώσεις χαρακτηριστικών:

```

class Weather:
    "Example with a mutable station identifier"

    def __init__(self, station_id):
        self.station_id = station_id

    def change_station(self, station_id):
        self.station_id = station_id

    def __eq__(self, other):
        return self.station_id == other.station_id

    def __hash__(self):
        return hash(self.station_id)

    @lru_cache(maxsize=20)
    def historic_rainfall(self, date, units='cm'):
        'Rainfall on a given date'
        # Depends on the station_id, date, and units.

```

## 2.7 Modules

### 2.7.1 Πως δημιουργώ ένα .pyc αρχείο;

Όταν ένα module εισάγεται για πρώτη φορά (ή όταν το αρχείο προέλευσης έχει αλλάξει από τη δημιουργία του τρέχοντος μεταγλωττισμένου αρχείου), ένα αρχείο `.pyc` που παρέχει τον μεταγλωττισμένο κώδικα θα πρέπει να δημιουργηθεί σε έναν υποκατάλογο `__pycache__` ο κατάλογος που περιέχει το `.py`. Το αρχείο `.pyc` θα έχει ένα όνομα αρχείου που ξεκινά με το ίδιο όνομα με το αρχείο `.py` και τελειώνει σε `.pyc`, με ένα μεσαίο στοιχείο που εξαρτάται από το συγκεκριμένο δυαδικό αρχείο python που το δημιούργησε. (Βλ. [PEP 3147](#) για λεπτομέρειες.)

Ένας λόγος για τον οποίο ενδέχεται να μην δημιουργηθεί ένα αρχείο `.pyc` είναι ένα πρόβλημα δικαιωμάτων στον κατάλογο που περιέχει το αρχείο προέλευσης, που σημαίνει ότι δεν μπορεί να δημιουργηθεί ο υποκατάλογος `__pycache__`. Αυτό μπορεί να συμβεί, για παράδειγμα, εάν αναπτυχθεί ως ένας χρήστης αλλά εκτελείται ως άλλος, όπως εάν δοκιμάζετε με έναν διακομιστή ιστού.

Εκτός και αν έχει οριστεί η μεταβλητή περιβάλλοντος `PYTHONDONTWRITEBYTECODE`, η δημιουργία ενός αρχείου `.pyc` είναι αυτόματη εάν εισάγετε ένα module και η Python έχει τη δυνατότητα (δικαιώματα, ελεύθερος χώρος, κ.λπ...) να δημιουργήσει ένα `__pycache__` υποκατάλογο και γράψει το μεταγλωττισμένο module σε αυτόν τον υποκατάλογο.

Η εκτέλεση της Python σε ένα σενάριο ανώτατου επιπέδου δεν θεωρείται εισαγωγή και δεν θα δημιουργηθεί `.pyc`. Για παράδειγμα, εάν έχετε ένα module ανωτάτου επιπέδου `foo.py` που εισάγει ένα άλλο module `xyz.py`, όταν εκτελείτε το `foo` (πληκτρολογώντας `python foo.py` ως εντολή κελύφους), θα δημιουργηθεί ένα `.pyc` για το `xyz` επειδή το `xyz` έχει εισαχθεί, αλλά δεν θα δημιουργηθεί αρχείο `.pyc` για το `foo` καθώς το `foo.py` δεν εισάγεται.

Εάν χρειάζεται να δημιουργήσετε ένα αρχείο `.pyc` για το `foo` – δηλαδή, να δημιουργήσετε ένα αρχείο `.pyc` για ένα module που δεν έχει εισαχθεί – μπορείτε, χρησιμοποιώντας τα modules `py_compile` και `compileall`.

Το module `py_compile` μπορεί να μεταγλωττίσει χειροκίνητα οποιαδήποτε module. Ένας τρόπος είναι να χρησιμοποιήσετε τη συνάρτηση `compile()` σε αυτήν την ενότητα διαδραστικά:

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

Αυτό θα γράψει το `.pyc` σε έναν υποκατάλογο `__pycache__` στην ίδια θέση με το `foo.py` (ή μπορείτε να το παρακάμψετε με την προαιρετική παράμετρο `cfile`).

Μπορείτε επίσης να μεταγλωττίσετε αυτόματα όλα τα αρχεία σε έναν κατάλογο ή καταλόγους χρησιμοποιώντας το module `compileall`. Μπορείτε να κάνετε από το shell prompt εκτελώντας το `compileall.py` και παρέχοντας τη διαδρομή ενός καταλόγου που περιέχει αρχεία Python για μεταγλώττιση:

```
python -m compileall .
```

## 2.7.2 Πως μπορώ να βρω το όνομα του τρέχοντος module;

Ένα module μπορεί να βρει το δικό του όνομα module κοιτάζοντας την προκαθορισμένη καθολική μεταβλητή `__name__`. Εάν αυτή έχει την τιμή `__main__`, το πρόγραμμα εκτελείται ως σενάριο. Πολλά modules που χρησιμοποιούνται συνήθως με την εισαγωγή τους παρέχουν επίσης μια διεπαφή γραμμής εντολών ή έναν αυτοέλεγχο και εκτελέστε αυτόν τον κώδικα μόνο αφού ελέγξετε το `__name__`:

```
def main():
    print('Running test...')
    ...

if __name__ == '__main__':
    main()
```

### 2.7.3 Πως μπορώ να έχω modules που εισάγουν αμοιβαία το ένα το άλλο;

Υποθέστε ότι έχετε τα ακόλουθα modules:

foo.py:

```
from bar import bar_var
foo_var = 1
```

bar.py:

```
from foo import foo_var
bar_var = 2
```

Το πρόβλημα είναι ότι ο διερμηνέας θα εκτελέσει τα ακόλουθα βήματα:

- main εισάγει foo
- Δημιουργούνται κενά καθολικά για το foo
- Το foo μεταγλωττίζεται και ξεκινά η εκτέλεση
- foo εισάγει bar
- Δημιουργούνται κενά καθολικά για bar
- Το bar μεταγλωττίζεται και αρχίζει να εκτελείται
- Το bar εισάγει το foo (το οποίο είναι απαγορευτικό, καθώς υπάρχει ήδη ένα module με το όνομα foo)
- Ο μηχανισμός εισαγωγής προσπαθεί να διαβάσει το foo\_var από τα παγκόσμια foo, για να ορίσει το bar.foo\_var = foo.foo\_var

Το τελευταίο βήμα αποτυγχάνει, επειδή η Python δεν έχει τελειώσει ακόμα με την ερμηνεία του foo και το global λεξικό συμβόλων για το foo είναι ακόμα κενό.

Το ίδιο συμβαίνει όταν χρησιμοποιείτε το `import foo` και, στη συνέχεια, προσπαθείτε να αποκτήσετε πρόσβαση στο `foo.foo_var` σε καθολικό κώδικα.

Υπάρχουν (τουλάχιστον) τρεις πιθανοί τρόποι αντιμετώπισης αυτού του προβλήματος.

Ο Guido van Rossum συνιστά την αποφυγή όλων των χρήσεων του `from <module> import ...` και την τοποθέτηση όλου του κώδικα μέσα σε συναρτήσεις. Τα initializations καθολικών μεταβλητών και μεταβλητών κλάσης θα πρέπει να χρησιμοποιηθούν μόνο σταθερές ή ενσωματωμένες συναρτήσεις. Αυτό σημαίνει ότι ένα εισαγόμενο module αναφέρεται ως `<module>.<name>`.

Ο Jim Roskind προτείνει να εκτελέσετε τα βήματα με την ακόλουθη σειρά σε κάθε module:

- εξαγωγές (globals, συναρτήσεις, και κλάσεις που δεν χρειάζονται εισαγόμενες βασικές κλάσεις)
- δηλώσεις `import`
- ενεργός κώδικας (συμπεριλαμβανομένων των καθολικών που αρχικοποιούνται από εισαγόμενες τιμές).

Ο Van Rossum δεν αρέσει πολύ αυτή η προσέγγιση επειδή οι εισαγωγές εμφανίζονται σε ένα περίεργο μέρος, αλλά λειτουργεί.

Ο Matthias Urlichs συνιστά την αναδιάρθρωση του κώδικά σας έτσι ώστε η αναδρομική εισαγωγή να μην είναι απαραίτητη εξ αρχής.

Αυτές οι λύσεις δεν αλληλοαποκλείονται.

### 2.7.4 `__import__`("x.y.z") επιστρέφει <module "x">• πως μπορώ να πάρω το z?

Σκεφτείτε να χρησιμοποιήσετε τη συνάρτηση ευκολίας `import_module()` από το `importlib` αντί:

```
z = importlib.import_module('x.y.z')
```

### 2.7.5 Όταν επεξεργάζομαι ένα module που έχει εισαχθεί και την επανεισάγω, οι αλλαγές δεν εμφανίζονται. Γιατί συμβαίνει αυτό;

Για λόγους αποτελεσματικότητας καθώς και συνέπειας, η Python διαβάζει το αρχείο της ενότητας μόνο την πρώτη φορά που εισάγεται μια λειτουργική μονάδα. Εάν δεν το έκανε, σε ένα πρόγραμμα που αποτελείται από πολλές ενότητες όπου η καθεμία εισάγει το ίδιο βασικό module, το βασικό module θα αναλυθεί και θα αναλυθεί ξανά πολλές φορές. Για να αναγκάσετε τη εκ νέου ανάγνωση μιας αλλαγμένης ενότητας, κάντε το εξής:

```
import importlib
import modname
importlib.reload(modname)
```

Προειδοποίηση: αυτή η τεχνική δεν είναι 100% ασφαλής. Ειδικότερα, modules που περιέχουν δηλώσεις όπως

```
from modname import some_objects
```

θα συνεχίσει να λειτουργεί με την παλιά έκδοση των εισαγόμενων αντικειμένων. Εάν η λειτουργική μονάδα περιέχει ορισμούς κλάσεων, οι υπάρχουσες παρουσίες κλάσεων δεν θα ενημερωθούν για να χρησιμοποιούν τον ορισμό της νέας κλάσης. Αυτό μπορεί να οδηγήσει στην ακόλουθη παράδοξη συμπεριφορά:

```
>>> import importlib
>>> import cls
>>> c = cls.C()                # Create an instance of C
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)      # isinstance is false?!?
False
```

Η φύση του προβλήματος καθίσταται σαφής εάν εκτυπώσετε την «ταυτότητα» των αντικειμένων κλάσης:

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```



### 3.1 Why does Python use indentation for grouping of statements?

Guido van Rossum believes that using indentation for grouping is extremely elegant and contributes a lot to the clarity of the average Python program. Most people learn to love this feature after a while.

Since there are no begin/end brackets there cannot be a disagreement between grouping perceived by the parser and the human reader. Occasionally C programmers will encounter a fragment of code like this:

```
if (x <= y)
    x++;
    y--;
z++;
```

Only the `x++` statement is executed if the condition is true, but the indentation leads many to believe otherwise. Even experienced C programmers will sometimes stare at it a long time wondering as to why `y` is being decremented even for `x > y`.

Because there are no begin/end brackets, Python is much less prone to coding-style conflicts. In C there are many different ways to place the braces. After becoming used to reading and writing code using a particular style, it is normal to feel somewhat uneasy when reading (or being required to write) in a different one.

Many coding styles place begin/end brackets on a line by themselves. This makes programs considerably longer and wastes valuable screen space, making it harder to get a good overview of a program. Ideally, a function should fit on one screen (say, 20–30 lines). 20 lines of Python can do a lot more work than 20 lines of C. This is not solely due to the lack of begin/end brackets – the lack of declarations and the high-level data types are also responsible – but the indentation-based syntax certainly helps.

## 3.2 Why am I getting strange results with simple arithmetic operations?

See the next question.

## 3.3 Why are floating-point calculations so inaccurate?

Users are often surprised by results like this:

```
>>> 1.2 - 1.0
0.19999999999999996
```

and think it is a bug in Python. It's not. This has little to do with Python, and much more to do with how the underlying platform handles floating-point numbers.

The `float` type in CPython uses a C `double` for storage. A `float` object's value is stored in binary floating-point with a fixed precision (typically 53 bits) and Python uses C operations, which in turn rely on the hardware implementation in the processor, to perform floating-point operations. This means that as far as floating-point operations are concerned, Python behaves like many popular languages including C and Java.

Many numbers that can be written easily in decimal notation cannot be expressed exactly in binary floating-point. For example, after:

```
>>> x = 1.2
```

the value stored for `x` is a (very good) approximation to the decimal value `1.2`, but is not exactly equal to it. On a typical machine, the actual stored value is:

```
1.0011001100110011001100110011001100110011001100110011001100110011 (binary)
```

which is exactly:

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

The typical precision of 53 bits provides Python floats with 15–16 decimal digits of accuracy.

For a fuller explanation, please see the floating point arithmetic chapter in the Python tutorial.

## 3.4 Why are Python strings immutable?

There are several advantages.

One is performance: knowing that a string is immutable means we can allocate space for it at creation time, and the storage requirements are fixed and unchanging. This is also one of the reasons for the distinction between tuples and lists.

Another advantage is that strings in Python are considered as «elemental» as numbers. No amount of activity will change the value `8` to anything else, and in Python, no amount of activity will change the string «eight» to anything else.



## 3.5 Why must “self” be used explicitly in method definitions and calls?

The idea was borrowed from Modula-3. It turns out to be very useful, for a variety of reasons.

First, it's more obvious that you are using a method or instance attribute instead of a local variable. Reading `self.x` or `self.meth()` makes it absolutely clear that an instance variable or method is used even if you don't know the class definition by heart. In C++, you can sort of tell by the lack of a local variable declaration (assuming globals are rare or easily recognizable) – but in Python, there are no local variable declarations, so you'd have to look up the class definition to be sure. Some C++ and Java coding standards call for instance attributes to have an `m_` prefix, so this explicitness is still useful in those languages, too.

Second, it means that no special syntax is necessary if you want to explicitly reference or call the method from a particular class. In C++, if you want to use a method from a base class which is overridden in a derived class, you have to use the `::` operator – in Python you can write `baseclass.methodname(self, <argument list>)`. This is particularly useful for `__init__()` methods, and in general in cases where a derived class method wants to extend the base class method of the same name and thus has to call the base class method somehow.

Finally, for instance variables it solves a syntactic problem with assignment: since local variables in Python are (by definition!) those variables to which a value is assigned in a function body (and that aren't explicitly declared global), there has to be some way to tell the interpreter that an assignment was meant to assign to an instance variable instead of to a local variable, and it should preferably be syntactic (for efficiency reasons). C++ does this through declarations, but Python doesn't have declarations and it would be a pity having to introduce them just for this purpose. Using the explicit `self.var` solves this nicely. Similarly, for using instance variables, having to write `self.var` means that references to unqualified names inside a method don't have to search the instance's directories. To put it another way, local variables and instance variables live in two different namespaces, and you need to tell Python which namespace to use.

## 3.6 Why can't I use an assignment in an expression?

Starting in Python 3.8, you can!

Assignment expressions using the walrus operator `:=` assign a variable in an expression:

```
while chunk := fp.read(200):
    print(chunk)
```

See [PEP 572](#) for more information.

## 3.7 Why does Python use methods for some functionality (e.g. `list.index()`) but functions for other (e.g. `len(list)`)?

As Guido said:

(a) For some operations, prefix notation just reads better than postfix – prefix (and infix!) operations have a long tradition in mathematics which likes notations where the visuals help the mathematician thinking about a problem. Compare the ease with which we rewrite a formula like  $x \cdot (a+b)$  into  $x \cdot a + x \cdot b$  to the clumsiness of doing the same thing using a raw OO notation.

(b) When I read code that says `len(x)` I *know* that it is asking for the length of something. This tells me two things: the result is an integer, and the argument is some kind of container. To the contrary, when I read `x.len()`, I have to already know that `x` is some kind of container implementing an interface or inheriting from a class that has a standard `len()`. Witness the confusion we occasionally have when a class that is not implementing a mapping has a `get()` or `keys()` method, or something that isn't a file has a `write()` method.

---<https://mail.python.org/pipermail/python-3000/2006-November/004643.html>

## 3.8 Why is join() a string method instead of a list or tuple method?

Strings became much more like other standard types starting in Python 1.6, when methods were added which give the same functionality that has always been available using the functions of the string module. Most of these new methods have been widely accepted, but the one which appears to make some programmers feel uncomfortable is:

```
"", ".join(['1', '2', '4', '8', '16'])
```

which gives the result:

```
"1, 2, 4, 8, 16"
```

There are two common arguments against this usage.

The first runs along the lines of: «It looks really ugly using a method of a string literal (string constant)», to which the answer is that it might, but a string literal is just a fixed value. If the methods are to be allowed on names bound to strings there is no logical reason to make them unavailable on literals.

The second objection is typically cast as: «I am really telling a sequence to join its members together with a string constant». Sadly, you aren't. For some reason there seems to be much less difficulty with having `split()` as a string method, since in that case it is easy to see that

```
"1, 2, 4, 8, 16".split(", ")
```

is an instruction to a string literal to return the substrings delimited by the given separator (or, by default, arbitrary runs of white space).

`join()` is a string method because in using it you are telling the separator string to iterate over a sequence of strings and insert itself between adjacent elements. This method can be used with any argument which obeys the rules for sequence objects, including any new classes you might define yourself. Similar methods exist for bytes and bytearray objects.

## 3.9 How fast are exceptions?

A try/except block is extremely efficient if no exceptions are raised. Actually catching an exception is expensive. In versions of Python prior to 2.0 it was common to use this idiom:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

This only made sense when you expected the dict to have the key almost all the time. If that wasn't the case, you coded it like this:

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

For this specific case, you could also use `value = dict.setdefault(key, getvalue(key))`, but only if the `getvalue()` call is cheap enough because it is evaluated in all cases.

## 3.10 Why isn't there a switch or case statement in Python?

In general, structured switch statements execute one block of code when an expression has a particular value or set of values. Since Python 3.10 one can easily match literal values, or constants within a namespace, with a `match ... case` statement. An older alternative is a sequence of `if... elif... elif... else`.

For cases where you need to choose from a very large number of possibilities, you can create a dictionary mapping case values to functions to call. For example:

```
functions = {'a': function_1,
            'b': function_2,
            'c': self.method_1}

func = functions[value]
func()
```

For calling methods on objects, you can simplify yet further by using the `getattr()` built-in to retrieve methods with a particular name:

```
class MyVisitor:
    def visit_a(self):
        ...

    def dispatch(self, value):
        method_name = 'visit_' + str(value)
        method = getattr(self, method_name)
        method()
```

It's suggested that you use a prefix for the method names, such as `visit_` in this example. Without such a prefix, if values are coming from an untrusted source, an attacker would be able to call any method on your object.

Imitating switch with fallthrough, as with C's switch-case-default, is possible, much harder, and less needed.

## 3.11 Can't you emulate threads in the interpreter instead of relying on an OS-specific thread implementation?

Answer 1: Unfortunately, the interpreter pushes at least one C stack frame for each Python stack frame. Also, extensions can call back into Python at almost random moments. Therefore, a complete threads implementation requires thread support for C.

Answer 2: Fortunately, there is [Stackless Python](#), which has a completely redesigned interpreter loop that avoids the C stack.

## 3.12 Why can't lambda expressions contain statements?

Python lambda expressions cannot contain statements because Python's syntactic framework can't handle statements nested inside expressions. However, in Python, this is not a serious problem. Unlike lambda forms in other languages, where they add functionality, Python lambdas are only a shorthand notation if you're too lazy to define a function.

Functions are already first class objects in Python, and can be declared in a local scope. Therefore the only advantage of using a lambda instead of a locally defined function is that you don't need to invent a name for the function – but that's just a local variable to which the function object (which is exactly the same type of object that a lambda expression yields) is assigned!

## 3.13 Can Python be compiled to machine code, C or some other language?

[Cython](#) compiles a modified version of Python with optional annotations into C extensions. [Nuitka](#) is an up-and-coming compiler of Python into C++ code, aiming to support the full Python language.

## 3.14 How does Python manage memory?

The details of Python memory management depend on the implementation. The standard implementation of Python, [CPython](#), uses reference counting to detect inaccessible objects, and another mechanism to collect reference cycles, periodically executing a cycle detection algorithm which looks for inaccessible cycles and deletes the objects involved. The `gc` module provides functions to perform a garbage collection, obtain debugging statistics, and tune the collector's parameters.

Other implementations (such as [Jython](#) or [PyPy](#)), however, can rely on a different mechanism such as a full-blown garbage collector. This difference can cause some subtle porting problems if your Python code depends on the behavior of the reference counting implementation.

In some Python implementations, the following code (which is fine in CPython) will probably run out of file descriptors:

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

Indeed, using CPython's reference counting and destructor scheme, each new assignment to `f` closes the previous file. With a traditional GC, however, those file objects will only get collected (and closed) at varying and possibly long intervals.

If you want to write code that will work with any Python implementation, you should explicitly close the file or use the `with` statement; this will work regardless of memory management scheme:

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

## 3.15 Why doesn't CPython use a more traditional garbage collection scheme?

For one thing, this is not a C standard feature and hence it's not portable. (Yes, we know about the Boehm GC library. It has bits of assembler code for *most* common platforms, not for all of them, and although it is mostly transparent, it isn't completely transparent; patches are required to get Python to work with it.)

Traditional GC also becomes a problem when Python is embedded into other applications. While in a standalone Python it's fine to replace the standard `malloc()` and `free()` with versions provided by the GC library, an application embedding Python may want to have its *own* substitute for `malloc()` and `free()`, and may not want Python's. Right now, CPython works with anything that implements `malloc()` and `free()` properly.

## 3.16 Why isn't all memory freed when CPython exits?

Objects referenced from the global namespaces of Python modules are not always deallocated when Python exits. This may happen if there are circular references. There are also certain bits of memory that are allocated by the C library that are impossible to free (e.g. a tool like Purify will complain about these). Python is, however, aggressive about cleaning up memory on exit and does try to destroy every single object.

If you want to force Python to delete certain things on deallocation use the `atexit` module to run a function that will force those deletions.

## 3.17 Why are there separate tuple and list data types?

Lists and tuples, while similar in many respects, are generally used in fundamentally different ways. Tuples can be thought of as being similar to Pascal records or C structs; they're small collections of related data which may be of different types which are operated on as a group. For example, a Cartesian coordinate is appropriately represented as a tuple of two or three numbers.

Lists, on the other hand, are more like arrays in other languages. They tend to hold a varying number of objects all of which have the same type and which are operated on one-by-one. For example, `os.listdir('.')` returns a list of strings representing the files in the current directory. Functions which operate on this output would generally not break if you added another file or two to the directory.

Tuples are immutable, meaning that once a tuple has been created, you can't replace any of its elements with a new value. Lists are mutable, meaning that you can always change a list's elements. Only immutable elements can be used as dictionary keys, and hence only tuples and not lists can be used as keys.

## 3.18 How are lists implemented in CPython?

CPython's lists are really variable-length arrays, not Lisp-style linked lists. The implementation uses a contiguous array of references to other objects, and keeps a pointer to this array and the array's length in a list head structure.

This makes indexing a list `a[i]` an operation whose cost is independent of the size of the list or the value of the index.

When items are appended or inserted, the array of references is resized. Some cleverness is applied to improve the performance of appending items repeatedly; when the array must be grown, some extra space is allocated so the next few times don't require an actual resize.

## 3.19 How are dictionaries implemented in CPython?

CPython's dictionaries are implemented as resizable hash tables. Compared to B-trees, this gives better performance for lookup (the most common operation by far) under most circumstances, and the implementation is simpler.

Dictionaries work by computing a hash code for each key stored in the dictionary using the `hash()` built-in function. The hash code varies widely depending on the key and a per-process seed; for example, «Python» could hash to -539294296 while «python», a string that differs by a single bit, could hash to 1142331976. The hash code is then used to calculate a location in an internal array where the value will be stored. Assuming that you're storing keys that all have different hash values, this means that dictionaries take constant time –  $O(1)$ , in Big-O notation – to retrieve a key.

## 3.20 Why must dictionary keys be immutable?

The hash table implementation of dictionaries uses a hash value calculated from the key value to find the key. If the key were a mutable object, its value could change, and thus its hash could also change. But since whoever changes the key object can't tell that it was being used as a dictionary key, it can't move the entry around in the dictionary. Then, when you try to look up the same object in the dictionary it won't be found because its hash value is different. If you tried to look up the old value it wouldn't be found either, because the value of the object found in that hash bin would be different.

If you want a dictionary indexed with a list, simply convert the list to a tuple first; the function `tuple(L)` creates a tuple with the same entries as the list `L`. Tuples are immutable and can therefore be used as dictionary keys.

Some unacceptable solutions that have been proposed:

- Hash lists by their address (object ID). This doesn't work because if you construct a new list with the same value it won't be found; e.g.:

```
mydict = {[1, 2]: '12'}
print(mydict[[1, 2]])
```

would raise a `KeyError` exception because the id of the `[1, 2]` used in the second line differs from that in the first line. In other words, dictionary keys should be compared using `==`, not using `is`.

- Make a copy when using a list as a key. This doesn't work because the list, being a mutable object, could contain a reference to itself, and then the copying code would run into an infinite loop.
- Allow lists as keys but tell the user not to modify them. This would allow a class of hard-to-track bugs in programs when you forgot or modified a list by accident. It also invalidates an important invariant of dictionaries: every value in `d.keys()` is usable as a key of the dictionary.
- Mark lists as read-only once they are used as a dictionary key. The problem is that it's not just the top-level object that could change its value; you could use a tuple containing a list as a key. Entering anything as a key into a dictionary would require marking all objects reachable from there as read-only – and again, self-referential objects could cause an infinite loop.

There is a trick to get around this if you need to, but use it at your own risk: You can wrap a mutable structure inside a class instance which has both a `__eq__()` and a `__hash__()` method. You must then make sure that the hash value for all such wrapper objects that reside in a dictionary (or other hash based structure), remain fixed while the object is in the dictionary (or other structure).

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 + i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

Note that the hash computation is complicated by the possibility that some members of the list may be unhashable and also by the possibility of arithmetic overflow.

Furthermore it must always be the case that if `o1 == o2` (ie `o1.__eq__(o2)` is True) then `hash(o1) == hash(o2)` (ie, `o1.__hash__() == o2.__hash__()`), regardless of whether the object is in a dictionary or not. If you fail to meet these restrictions dictionaries and other hash based structures will misbehave.

In the case of `ListWrapper`, whenever the wrapper object is in a dictionary the wrapped list must not change to avoid anomalies. Don't do this unless you are prepared to think hard about the requirements and the consequences of not meeting them correctly. Consider yourself warned.

## 3.21 Why doesn't `list.sort()` return the sorted list?

In situations where performance matters, making a copy of the list just to sort it would be wasteful. Therefore, `list.sort()` sorts the list in place. In order to remind you of that fact, it does not return the sorted list. This way, you won't be fooled into accidentally overwriting a list when you need a sorted copy but also need to keep the unsorted version around.

If you want to return a new list, use the built-in `sorted()` function instead. This function creates a new list from a provided iterable, sorts it and returns it. For example, here's how to iterate over the keys of a dictionary in sorted order:

```
for key in sorted(mydict):
    ... # do whatever with mydict[key]...
```

## 3.22 How do you specify and enforce an interface spec in Python?

An interface specification for a module as provided by languages such as C++ and Java describes the prototypes for the methods and functions of the module. Many feel that compile-time enforcement of interface specifications helps in the construction of large programs.

Python 2.6 adds an `abc` module that lets you define Abstract Base Classes (ABCs). You can then use `isinstance()` and `issubclass()` to check whether an instance or a class implements a particular ABC. The `collections.abc` module defines a set of useful ABCs such as `Iterable`, `Container`, and `MutableMapping`.

For Python, many of the advantages of interface specifications can be obtained by an appropriate test discipline for components.

A good test suite for a module can both provide a regression test and serve as a module interface specification and a set of examples. Many Python modules can be run as a script to provide a simple «self test.» Even modules which use complex external interfaces can often be tested in isolation using trivial «stub» emulations of the external interface. The `doctest` and `unittest` modules or third-party test frameworks can be used to construct exhaustive test suites that exercise every line of code in a module.

An appropriate testing discipline can help build large complex applications in Python as well as having interface specifications would. In fact, it can be better because an interface specification cannot test certain properties of a program. For example, the `list.append()` method is expected to add new elements to the end of some internal list; an interface specification cannot test that your `list.append()` implementation will actually do this correctly, but it's trivial to check this property in a test suite.

Writing test suites is very helpful, and you might want to design your code to make it easily tested. One increasingly popular technique, test-driven development, calls for writing parts of the test suite first, before you write any of the actual code. Of course Python allows you to be sloppy and not write test cases at all.

## 3.23 Why is there no goto?

In the 1970s people realized that unrestricted goto could lead to messy «spaghetti» code that was hard to understand and revise. In a high-level language, it is also unneeded as long as there are ways to branch (in Python, with `if` statements and `or`, `and`, and `if-else` expressions) and loop (with `while` and `for` statements, possibly containing `continue` and `break`).

One can also use exceptions to provide a «structured goto» that works even across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the «go» or «goto» constructs of C, Fortran, and other languages. For example:

```
class label(Exception): pass # declare a label

try:
    ...
    if condition: raise label() # goto label
    ...
except label: # where to goto
    pass
...
```

This doesn't allow you to jump into the middle of a loop, but that's usually considered an abuse of goto anyway. Use sparingly.

## 3.24 Why can't raw strings (r-strings) end with a backslash?

More precisely, they can't end with an odd number of backslashes: the unpaired backslash at the end escapes the closing quote character, leaving an unterminated string.

Raw strings were designed to ease creating input for processors (chiefly regular expression engines) that want to do their own backslash escape processing. Such processors consider an unmatched trailing backslash to be an error anyway, so raw strings disallow that. In return, they allow you to pass on the string quote character by escaping it with a backslash. These rules work well when r-strings are used for their intended purpose.

If you're trying to build Windows pathnames, note that all Windows system calls accept forward slashes too:

```
f = open("/mydir/file.txt") # works fine!
```

If you're trying to build a pathname for a DOS command, try e.g. one of

```
dir = r"\this\is\my\dos\dir" "\\"
dir = r"\this\is\my\dos\dir\" "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```



## 3.25 Why doesn't Python have a «with» statement for attribute assignments?

Python has a “with” statement that wraps the execution of a block, calling code on the entrance and exit from the block. Some languages have a construct that looks like this:

```
with obj:
    a = 1          # equivalent to obj.a = 1
    total = total + 1  # obj.total = obj.total + 1
```

In Python, such a construct would be ambiguous.

Other languages, such as Object Pascal, Delphi, and C++, use static types, so it's possible to know, in an unambiguous way, what member is being assigned to. This is the main point of static typing – the compiler *always* knows the scope of every variable at compile time.

Python uses dynamic types. It is impossible to know in advance which attribute will be referenced at runtime. Member attributes may be added or removed from objects on the fly. This makes it impossible to know, from a simple reading, what attribute is being referenced: a local one, a global one, or a member attribute?

For instance, take the following incomplete snippet:

```
def foo(a):
    with a:
        print(x)
```

The snippet assumes that «a» must have a member attribute called «x». However, there is nothing in Python that tells the interpreter this. What should happen if «a» is, let us say, an integer? If there is a global variable named «x», will it be used inside the with block? As you see, the dynamic nature of Python makes such choices much harder.

The primary benefit of «with» and similar language features (reduction of code volume) can, however, easily be achieved in Python by assignment. Instead of:

```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

write this:

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

This also has the side-effect of increasing execution speed because name bindings are resolved at run-time in Python, and the second version only needs to perform the resolution once.

## 3.26 Why don't generators support the with statement?

For technical reasons, a generator used directly as a context manager would not work correctly. When, as is most common, a generator is used as an iterator run to completion, no closing is needed. When it is, wrap it as `contextlib.closing(generator)` in the “with” statement.

## 3.27 Why are colons required for the if/while/def/class statements?

The colon is required primarily to enhance readability (one of the results of the experimental ABC language). Consider this:

```
if a == b
    print(a)
```

versus

```
if a == b:
    print(a)
```

Notice how the second one is slightly easier to read. Notice further how a colon sets off the example in this FAQ answer; it's a standard usage in English.

Another minor reason is that the colon makes it easier for editors with syntax highlighting; they can look for colons to decide when indentation needs to be increased instead of having to do a more elaborate parsing of the program text.

## 3.28 Why does Python allow commas at the end of lists and tuples?

Python lets you add a trailing comma at the end of lists, tuples, and dictionaries:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional but good style
}
```

There are several reasons to allow this.

When you have a literal value for a list, tuple, or dictionary spread across multiple lines, it's easier to add more elements because you don't have to remember to add a comma to the previous line. The lines can also be reordered without creating a syntax error.

Accidentally omitting the comma can lead to errors that are hard to diagnose. For example:

```
x = [
    "fee",
    "fie",
    "foo",
    "fum"
]
```

This list looks like it has four elements, but it actually contains three: «fee», «fiefoo» and «fum». Always adding the comma avoids this source of error.

Allowing the trailing comma may also make programmatic code generation easier.



---

## Library and Extension FAQ

---

### 4.1 General Library Questions

#### 4.1.1 How do I find a module or application to perform task X?

Check the Library Reference to see if there's a relevant standard library module. (Eventually you'll learn what's in the standard library and will be able to skip this step.)

For third-party packages, search the [Python Package Index](#) or try [Google](#) or another web search engine. Searching for «Python» plus a keyword or two for your topic of interest will usually find something helpful.

#### 4.1.2 Where is the `math.py` (`socket.py`, `regex.py`, etc.) source file?

If you can't find a source file for a module it may be a built-in or dynamically loaded module implemented in C, C++ or other compiled language. In this case you may not have the source file or it may be something like `mathmodule.c`, somewhere in a C source directory (not on the Python Path).

There are (at least) three kinds of modules in Python:

- 1) modules written in Python (`.py`);
- 2) modules written in C and dynamically loaded (`.dll`, `.pyd`, `.so`, `.sl`, etc);
- 3) modules written in C and linked with the interpreter; to get a list of these, type:

```
import sys
print(sys.builtin_module_names)
```

### 4.1.3 How do I make a Python script executable on Unix?

You need to do two things: the script file's mode must be executable and the first line must begin with `#!` followed by the path of the Python interpreter.

The first is done by executing `chmod +x scriptfile` or perhaps `chmod 755 scriptfile`.

The second can be done in a number of ways. The most straightforward way is to write

```
#!/usr/local/bin/python
```

as the very first line of your file, using the pathname for where the Python interpreter is installed on your platform.

If you would like the script to be independent of where the Python interpreter lives, you can use the `env` program. Almost all Unix variants support the following, assuming the Python interpreter is in a directory on the user's `PATH`:

```
#!/usr/bin/env python
```

*Don't* do this for CGI scripts. The `PATH` variable for CGI scripts is often very minimal, so you need to use the actual absolute pathname of the interpreter.

Occasionally, a user's environment is so full that the `/usr/bin/env` program fails; or there's no `env` program at all. In that case, you can try the following hack (due to Alex Rezinsky):

```
#!/bin/sh
""" : """
exec python $0 ${1+"$@"}
"""
```

The minor disadvantage is that this defines the script's `__doc__` string. However, you can fix that by adding

```
__doc__ = """...Whatever..."""
```

### 4.1.4 Is there a curses/termcap package for Python?

For Unix variants: The standard Python source distribution comes with a `curses` module in the `Modules` subdirectory, though it's not compiled by default. (Note that this is not available in the Windows distribution – there is no `curses` module for Windows.)

The `curses` module supports basic curses features as well as many additional functions from `ncurses` and `SVSV curses` such as colour, alternative character set support, pads, and mouse support. This means the module isn't compatible with operating systems that only have BSD curses, but there don't seem to be any currently maintained OSes that fall into this category.

### 4.1.5 Is there an equivalent to C's `onexit()` in Python?

The `atexit` module provides a register function that is similar to C's `onexit()`.

### 4.1.6 Why don't my signal handlers work?

The most common problem is that the signal handler is declared with the wrong argument list. It is called as

```
handler(signum, frame)
```

so it should be declared with two parameters:

```
def handler(signum, frame):
    ...
```

## 4.2 Common tasks

### 4.2.1 How do I test a Python program or component?

Python comes with two testing frameworks. The `doctest` module finds examples in the docstrings for a module and runs them, comparing the output with the expected output given in the docstring.

The `unittest` module is a fancier testing framework modelled on Java and Smalltalk testing frameworks.

To make testing easier, you should use good modular design in your program. Your program should have almost all functionality encapsulated in either functions or class methods – and this sometimes has the surprising and delightful effect of making the program run faster (because local variable accesses are faster than global accesses). Furthermore the program should avoid depending on mutating global variables, since this makes testing much more difficult to do.

The «global main logic» of your program may be as simple as

```
if __name__ == "__main__":
    main_logic()
```

at the bottom of the main module of your program.

Once your program is organized as a tractable collection of function and class behaviours, you should write test functions that exercise the behaviours. A test suite that automates a sequence of tests can be associated with each module. This sounds like a lot of work, but since Python is so terse and flexible it's surprisingly easy. You can make coding much more pleasant and fun by writing your test functions in parallel with the «production code», since this makes it easy to find bugs and even design flaws earlier.

«Support modules» that are not intended to be the main module of a program may include a self-test of the module.

```
if __name__ == "__main__":
    self_test()
```

Even programs that interact with complex external interfaces may be tested when the external interfaces are unavailable by using «fake» interfaces implemented in Python.

## 4.2.2 How do I create documentation from doc strings?

The `pydoc` module can create HTML from the doc strings in your Python source code. An alternative for creating API documentation purely from docstrings is `epydoc`. `Sphinx` can also include docstring content.

## 4.2.3 How do I get a single keypress at a time?

For Unix variants there are several solutions. It's straightforward to do this using `curses`, but `curses` is a fairly large module to learn.

# 4.3 Threads

## 4.3.1 How do I program using threads?

Be sure to use the `threading` module and not the `_thread` module. The `threading` module builds convenient abstractions on top of the low-level primitives provided by the `_thread` module.

## 4.3.2 None of my threads seem to run: why?

As soon as the main thread exits, all threads are killed. Your main thread is running too quickly, giving the threads no time to do any work.

A simple fix is to add a sleep to the end of the program that's long enough for all the threads to finish:

```
import threading, time

def thread_task(name, n):
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10) # <-----! 
```

But now (on many platforms) the threads don't run in parallel, but appear to run sequentially, one at a time! The reason is that the OS thread scheduler doesn't start a new thread until the previous thread is blocked.

A simple fix is to add a tiny sleep to the start of the run function:

```
def thread_task(name, n):
    time.sleep(0.001) # <-----!
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```



Instead of trying to guess a good delay value for `time.sleep()`, it's better to use some kind of semaphore mechanism. One idea is to use the `queue` module to create a queue object, let each thread append a token to the queue when it finishes, and let the main thread read as many tokens from the queue as there are threads.

### 4.3.3 How do I parcel out work among a bunch of worker threads?

The easiest way is to use the `concurrent.futures` module, especially the `ThreadPoolExecutor` class.

Or, if you want fine control over the dispatching algorithm, you can write your own logic manually. Use the `queue` module to create a queue containing a list of jobs. The `Queue` class maintains a list of objects and has a `.put(obj)` method that adds items to the queue and a `.get()` method to return them. The class will take care of the locking necessary to ensure that each job is handed out exactly once.

Here's a trivial example:

```
import threading, queue, time

# The worker thread gets jobs off the queue. When the queue is empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print('Running worker')
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.current_thread(), end=' ')
            print('queue empty')
            break
        else:
            print('Worker', threading.current_thread(), end=' ')
            print('running with argument', arg)
            time.sleep(0.5)

# Create queue
q = queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()

# Begin adding work to the queue
for i in range(50):
    q.put(i)

# Give threads time to run
print('Main thread sleeping')
time.sleep(5)
```

When run, this will produce the following output:

```
Running worker
Running worker
Running worker
Running worker
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with argument 0
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

Consult the module's documentation for more details; the `Queue` class provides a featureful interface.

#### 4.3.4 What kinds of global value mutation are thread-safe?

A *global interpreter lock* (GIL) is used internally to ensure that only one thread runs in the Python VM at a time. In general, Python offers to switch among threads only between bytecode instructions; how frequently it switches can be set via `sys.setswitchinterval()`. Each bytecode instruction and therefore all the C implementation code reached from each instruction is therefore atomic from the point of view of a Python program.

In theory, this means an exact accounting requires an exact understanding of the PVM bytecode implementation. In practice, it means that operations on shared variables of built-in data types (ints, lists, dicts, etc) that «look atomic» really are.

For example, the following operations are all atomic (L, L1, L2 are lists, D, D1, D2 are dicts, x, y are objects, i, j are ints):

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

These aren't:

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

Operations that replace other objects may invoke those other objects' `__del__()` method when their reference count reaches zero, and that can affect things. This is especially true for the mass updates to dictionaries and lists. When in doubt, use a mutex!

### 4.3.5 Can't we get rid of the Global Interpreter Lock?

The *global interpreter lock* (GIL) is often seen as a hindrance to Python's deployment on high-end multiprocessor server machines, because a multi-threaded Python program effectively only uses one CPU, due to the insistence that (almost) all Python code can only run while the GIL is held.

Back in the days of Python 1.5, Greg Stein actually implemented a comprehensive patch set (the «free threading» patches) that removed the GIL and replaced it with fine-grained locking. Adam Olsen recently did a similar experiment in his `python-safethread` project. Unfortunately, both experiments exhibited a sharp drop in single-thread performance (at least 30% slower), due to the amount of fine-grained locking necessary to compensate for the removal of the GIL.

This doesn't mean that you can't make good use of Python on multi-CPU machines! You just have to be creative with dividing the work up between multiple *processes* rather than multiple *threads*. The `ProcessPoolExecutor` class in the new `concurrent.futures` module provides an easy way of doing so; the `multiprocessing` module provides a lower-level API in case you want more control over dispatching of tasks.

Judicious use of C extensions will also help; if you use a C extension to perform a time-consuming task, the extension can release the GIL while the thread of execution is in the C code and allow other threads to get some work done. Some standard library modules such as `zlib` and `hashlib` already do this.

It has been suggested that the GIL should be a per-interpreter-state lock rather than truly global; interpreters then wouldn't be able to share objects. Unfortunately, this isn't likely to happen either. It would be a tremendous amount of work, because many object implementations currently have global state. For example, small integers and short strings are cached; these caches would have to be moved to the interpreter state. Other object types have their own free list; these free lists would have to be moved to the interpreter state. And so on.

And I doubt that it can even be done in finite time, because the same problem exists for 3rd party extensions. It is likely that 3rd party extensions are being written at a faster rate than you can convert them to store all their global state in the interpreter state.

And finally, once you have multiple interpreters not sharing any state, what have you gained over running each interpreter in a separate process?

## 4.4 Input and Output

### 4.4.1 How do I delete a file? (And other file questions...)

Use `os.remove(filename)` or `os.unlink(filename)`; for documentation, see the `os` module. The two functions are identical; `unlink()` is simply the name of the Unix system call for this function.

To remove a directory, use `os.rmdir()`; use `os.mkdir()` to create one. `os.makedirs(path)` will create any intermediate directories in `path` that don't exist. `os.removedirs(path)` will remove intermediate directories as long as they're empty; if you want to delete an entire directory tree and its contents, use `shutil.rmtree()`.

To rename a file, use `os.rename(old_path, new_path)`.

To truncate a file, open it using `f = open(filename, "rb+")`, and use `f.truncate(offset)`; `offset` defaults to the current seek position. There's also `os.ftruncate(fd, offset)` for files opened with `os.open()`, where `fd` is the file descriptor (a small integer).

The `shutil` module also contains a number of functions to work on files including `copyfile()`, `copytree()`, and `rmtree()`.

### 4.4.2 How do I copy a file?

The `shutil` module contains a `copyfile()` function. Note that on Windows NTFS volumes, it does not copy *alternate data streams* nor *resource forks* on macOS HFS+ volumes, though both are now rarely used. It also doesn't copy file permissions and metadata, though using `shutil.copy2()` instead will preserve most (though not all) of it.

### 4.4.3 How do I read (or write) binary data?

To read or write complex binary data formats, it's best to use the `struct` module. It allows you to take a string containing binary data (usually numbers) and convert it to Python objects; and vice versa.

For example, the following code reads two 2-byte integers and one 4-byte integer in big-endian format from a file:

```
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhI", s)
```

The “>” in the format string forces big-endian data; the letter “h” reads one «short integer» (2 bytes), and “I” reads one «long integer» (4 bytes) from the string.

For data that is more regular (e.g. a homogeneous list of ints or floats), you can also use the `array` module.

---

**Σημείωση:** To read and write binary data, it is mandatory to open the file in binary mode (here, passing “rb” to `open()`). If you use “r” instead (the default), the file will be open in text mode and `f.read()` will return `str` objects rather than `bytes` objects.

---

### 4.4.4 I can't seem to use `os.read()` on a pipe created with `os.popen()`; why?

`os.read()` is a low-level function which takes a file descriptor, a small integer representing the opened file. `os.popen()` creates a high-level file object, the same type returned by the built-in `open()` function. Thus, to read *n* bytes from a pipe *p* created with `os.popen()`, you need to use `p.read(n)`.

### 4.4.5 How do I access the serial (RS232) port?

For Win32, OSX, Linux, BSD, Jython, IronPython:

<https://pypi.org/project/pyserial/>

For Unix, see a Usenet post by Mitch Chapman:

<https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com>

#### 4.4.6 Why doesn't closing `sys.stdout` (`stdin`, `stderr`) really close it?

Python *file objects* are a high-level layer of abstraction on low-level C file descriptors.

For most file objects you create in Python via the built-in `open()` function, `f.close()` marks the Python file object as being closed from Python's point of view, and also arranges to close the underlying C file descriptor. This also happens automatically in `f`'s destructor, when `f` becomes garbage.

But `stdin`, `stdout` and `stderr` are treated specially by Python, because of the special status also given to them by C. Running `sys.stdout.close()` marks the Python-level file object as being closed, but does *not* close the associated C file descriptor.

To close the underlying C file descriptor for one of these three, you should first be sure that's what you really want to do (e.g., you may confuse extension modules trying to do I/O). If it is, use `os.close()`:

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

Or you can use the numeric constants 0, 1 and 2, respectively.

### 4.5 Network/Internet Programming

#### 4.5.1 What WWW tools are there for Python?

See the chapters titled `internet` and `netdata` in the Library Reference Manual. Python has many modules that will help you build server-side and client-side web systems.

A summary of available frameworks is maintained by Paul Boddie at <https://wiki.python.org/moin/WebProgramming>.

Cameron Laird maintains a useful set of pages about Python web technologies at [https://web.archive.org/web/20210224183619/http://phaseit.net/claird/comp.lang.python/web\\_python](https://web.archive.org/web/20210224183619/http://phaseit.net/claird/comp.lang.python/web_python).

#### 4.5.2 How can I mimic CGI form submission (METHOD=POST)?

I would like to retrieve web pages that are the result of POSTing a form. Is there existing code that would let me do this easily?

Yes. Here's a simple example that uses `urllib.request`:

```
#!/usr/local/bin/python

import urllib.request

# build the query string
qs = "First=Josephine&MI=Q&Last=Public"

# connect and send the server a path
req = urllib.request.urlopen('http://www.some-server.out-there'
                              '/cgi-bin/some-cgi-script', data=qs)

with req:
    msg, hdrs = req.read(), req.info()
```

Note that in general for percent-encoded POST operations, query strings must be quoted using `urllib.parse.urlencode()`. For example, to send `name=Guy Steele, Jr.`:

```
>>> import urllib.parse
>>> urllib.parse.urlencode({'name': 'Guy Steele, Jr.'})
'name=Guy+Steele%2C+Jr.'
```

**Δείτε επίσης:**

`urllib-howto` for extensive examples.

### 4.5.3 What module should I use to help with generating HTML?

You can find a collection of useful links on the [Web Programming wiki](#) page.

### 4.5.4 How do I send mail from a Python script?

Use the standard library module `smtplib`.

Here's a very simple interactive mail sender that uses it. This method will work on any host that supports an SMTP listener.

```
import sys, smtplib

fromaddr = input("From: ")
toaddrs = input("To: ").split(',')
print("Enter message, end with ^D:")
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
    msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

A Unix-only alternative uses `sendmail`. The location of the `sendmail` program varies between systems; sometimes it is `/usr/lib/sendmail`, sometimes `/usr/sbin/sendmail`. The `sendmail` manual page will help you out. Here's some sample code:

```
import os

SENDMAIL = "/usr/sbin/sendmail" # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n") # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

### 4.5.5 How do I avoid blocking in the connect() method of a socket?

The `select` module is commonly used to help with asynchronous I/O on sockets.

To prevent the TCP connect from blocking, you can set the socket to non-blocking mode. Then when you do the `connect()`, you will either connect immediately (unlikely) or get an exception that contains the error number as `.errno.errno.EINPROGRESS` indicates that the connection is in progress, but hasn't finished yet. Different OSes will return different values, so you're going to have to check what's returned on your system.

You can use the `connect_ex()` method to avoid creating an exception. It will just return the `errno` value. To poll, you can call `connect_ex()` again later – 0 or `errno.EISCONN` indicate that you're connected – or you can pass this socket to `select.select()` to check if it's writable.

---

**Σημείωση:** The `asyncio` module provides a general purpose single-threaded and concurrent asynchronous library, which can be used for writing non-blocking network code. The third-party [Twisted](#) library is a popular and feature-rich alternative.

---

## 4.6 Databases

### 4.6.1 Are there any interfaces to database packages in Python?

Yes.

Interfaces to disk-based hashes such as DBM and GDBM are also included with standard Python. There is also the `sqlite3` module, which provides a lightweight disk-based relational database.

Support for most relational databases is available. See the [DatabaseProgramming wiki page](#) for details.

### 4.6.2 How do you implement persistent objects in Python?

The `pickle` library module solves this in a very general way (though you still can't store things like open files, sockets or windows), and the `shelve` library module uses `pickle` and (g)dbm to create persistent mappings containing arbitrary Python objects.

## 4.7 Mathematics and Numerics

### 4.7.1 How do I generate random numbers in Python?

The standard module `random` implements a random number generator. Usage is simple:

```
import random
random.random()
```

This returns a random floating point number in the range [0, 1).

There are also many other specialized generators in this module, such as:

- `randrange(a, b)` chooses an integer in the range [a, b).
- `uniform(a, b)` chooses a floating point number in the range [a, b).
- `normalvariate(mean, sdev)` samples the normal (Gaussian) distribution.

Some higher-level functions operate on sequences directly, such as:

- `choice(S)` chooses a random element from a given sequence.
- `shuffle(L)` shuffles a list in-place, i.e. permutes it randomly.

There's also a `Random` class you can instantiate to create independent multiple random number generators.



---

## Συχνές ερωτήσεις επέκτασης/ενσωμάτωσης

---

### 5.1 Μπορώ να δημιουργήσω τις δικές μου συναρτήσεις στη C;

Ναι, μπορείτε να δημιουργήσετε ενσωματωμένα (built-in) modules που περιέχουν συναρτήσεις, μεταβλητές, εξαιρέσεις και ακόμη και νέους τύπους στην C. Αυτό εξηγείται στο έγγραφο `extending-index`.

Τα περισσότερα βιβλία μεσαίας ή προηγμένης Python θα καλύπτουν επίσης αυτό το θέμα.

### 5.2 Μπορώ να δημιουργήσω τις δικές μου συναρτήσεις στη C++;

Ναι, χρησιμοποιώντας τις δυνατότητες συμβατότητας C που βρίσκονται στη C++. Τοποθετήστε το `extern "C"` { ... } γύρω από την Python να περιλαμβάνει αρχεία και τοποθετήστε το `extern "C"` πριν από κάθε συνάρτηση που πρόκειται να κληθεί από τον διερμηνέα της Python. Τα καθολικά ή τα στατικά αντικείμενα C++ με constructors μάλλον δεν είναι καλή ιδέα.

### 5.3 Το να γράψει C κάποιος είναι δύσκολο· υπάρχουν άλλες εναλλακτικές;

Υπάρχουν διάφορες εναλλακτικές λύσεις για να γράψετε τις δικές σας επεκτάσεις C, ανάλογα με το τι προσπαθείτε να κάνετε.

Το `Cython` και το σχετικό του `Pyrex` είναι μεταγλωττιστές που δέχονται ελαφρώς τροποποιημένη μορφή της Python και δημιουργούν τον αντίστοιχο C κώδικα. Το `Cython` και το `Pyrex` καθιστούν δυνατή τη σύνταξη μιας επέκτασης χωρίς να χρειάζεται να μάθετε το C API της Python.

Εάν χρειάζεται να συνδεθείτε με κάποια βιβλιοθήκη C ή C++ για την οποία δεν υπάρχει αυτήν τη στιγμή επέκταση Python, μπορείτε να δοκιμάσετε να αναδιπλώσετε τους τύπους δεδομένων και τις συναρτήσεις της βιβλιοθήκης με ένα εργαλείο όπως `SWIG`, `SIP`, `CXX Boost`, ή `Weave` είναι επίσης εναλλακτικές λύσεις για την αναδίπλωση βιβλιοθηκών C++.

## 5.4 Πως μπορώ να εκτελέσω αυθαίρετες δηλώσεις Python από το C;

Η συνάρτηση υψηλότερου επιπέδου για να γίνει αυτό είναι η `PyRun_SimpleString()` η οποία εκτελεί ένα όρισμα συμβολοσειράς στο πλαίσιο της ενότητας `__main__` και επιστρέφει 0 για επιτυχία και -1 όταν συμβαίνει μια εξαίρεση (συμπεριλαμβανομένου του `SyntaxError`). Εάν θέλετε περισσότερο έλεγχο, χρησιμοποιήστε `PyRun_String()`· δείτε τον πηγαίο κώδικα `PyRun_SimpleString()` στο `Python/pythonrun.c`.

## 5.5 Πώς μπορώ να αξιολογήσω μια αυθαίρετη έκφραση Python από τη C;

Καλέστε τη συνάρτηση `PyRun_String()` από την προηγούμενη ερώτηση με το σύμβολο έναρξης `Py_eval_input`: αναλύει μια παράσταση, την αξιολογεί και επιστρέφει την τιμή της.

## 5.6 Πως μπορώ να εξάγω τιμές C από ένα αντικείμενο Python;

Αυτό εξαρτάται από τον τύπο του αντικειμένου. Εάν είναι μια πλειάδα (tuple), `PyTuple_Size()` επιστρέφει το μήκος του και το `PyTuple_GetItem()` επιστρέφει το στοιχείο σε ένα καθορισμένο index. Οι λίστες έχουν παρόμοιες συναρτήσεις, `PyList_Size()` και `PyList_GetItem()`.

Για bytes, `PyBytes_Size()` επιστρέφει το μήκος του και το `PyBytes_AsStringAndSize()` παρέχει έναν δείκτη στην τιμή και το μήκος του. Λάβετε υπόψη ότι τα αντικείμενα byte της Python μπορεί να περιέχουν null byte, επομένως η `strlen()` της C δεν πρέπει να χρησιμοποιείται.

Για να ελέγξετε τον τύπο ενός αντικειμένου, πρώτα βεβαιωθείτε ότι δεν είναι NULL, και μετά χρησιμοποιήστε τα `PyBytes_Check()`, `PyTuple_Check()`, `PyList_Check()`, κλπ.

Υπάρχει επίσης ένα API υψηλού επιπέδου για αντικείμενα Python που παρέχεται από τη λεγόμενη “abstract” διεπαφή (interface) – διαβάστε `Include/abstract.h` για περισσότερες λεπτομέρειες. Επιτρέπει τη διασύνδεση με κάθε είδους ακολουθίας Python χρησιμοποιώντας κλήσεις όπως `PySequence_Length()`, `PySequence_GetItem()`, κλπ. καθώς και πολλά άλλα χρήσιμα πρωτόκολλα όπως αριθμοί (`PyNumber_Index()` et al.) και αντιστοιχίσεις στον PyMapping APIs.

## 5.7 Πώς μπορώ να χρησιμοποιήσω την `Py_BuildValue()` για να δημιουργήσω μια πλειάδα (tuple) αυθαίρετου μήκους;

Δεν μπορείτε. Χρησιμοποιήστε το `PyTuple_Pack()`.

## 5.8 Πώς καλώ τη μέθοδο ενός αντικειμένου από τη C;

Η συνάρτηση `PyObject_CallMethod()` μπορεί να χρησιμοποιηθεί για την κλήση μιας αυθαίρετης μεθόδου ενός αντικειμένου. Οι παράμετροι είναι το αντικείμενο, το όνομα της μεθόδου προς κλήση, μια συμβολοσειρά μορφής όπως αυτή που χρησιμοποιείται με τη `Py_BuildValue()`, και τις τιμές ορίσματος:

```
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                    const char *arg_format, ...);
```

Αυτό λειτουργεί για κάθε αντικείμενο που έχει μεθόδους – είτε είναι ενσωματωμένες είτε καθορίζονται από το χρήστη. Είστε υπεύθυνοι εάν τελικά χρησιμοποιήσετε `Py_DECREF()` στην τιμή επιστροφής.

Για να καλέσετε, π.χ., τη μέθοδο «seek» ενός αντικειμένου αρχείου με ορίσματα 10, 0 (υποθέτοντας ότι ο δείκτης του αντικειμένου αρχείου είναι «f»):

```
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
    ... an exception occurred ...
}
else {
    Py_DECREF(res);
}
```

Σημειώστε ότι επειδή το `PyObject_CallObject()` πάντα θέλει μια πλειάδα (tuple) για τη λίστα ορισμάτων, για να καλέσει μια συνάρτηση χωρίς ορίσματα, να περάσει «()» για τη μορφή και να καλέσει μια συνάρτηση με ένα όρισμα, περιβάλλουν το όρισμα σε παρένθεση, π.χ. «(i)».

## 5.9 Πώς μπορώ να κάνω catch την έξοδο από την `PyErr_Print()` (ή οτιδήποτε εκτυπώνεται σε `stdout/stderr`);

Στον κώδικα Python, ορίστε ένα αντικείμενο που υποστηρίζει τη μέθοδο `write()`. Αντιστοιχίστε αυτό το αντικείμενο στα `sys.stdout` και `sys.stderr`. Καλέστε το `print_error`, ή απλώς επιτρέψτε στον τυπικό μηχανισμό ανίχνευσης να λειτουργήσει. Στη συνέχεια, η έξοδος θα πάει οπουδήποτε την στείλει η μέθοδος `write()`.

Ο ευκολότερος τρόπος για να το κάνετε αυτό είναι να χρησιμοποιήσετε την κλάση `io.StringIO`:

```
>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!
```

Ένα προσαρμοσμένο αντικείμενο για να κάνει το ίδιο θα μοιάζει με αυτό:

```
>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
... 
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write('.'.join(sys.stdout.data))
foo
hello world!
```

## 5.10 Πως μπορώ να αποκτήσω πρόσβαση σε ένα module γραμμένο σε Python από τη C;

Μπορείτε να λάβετε έναν δείκτη στο αντικείμενο του module ως εξής:

```
module = PyImport_ImportModule("<modulename>");
```

Εάν το module δεν έχει εισαχθεί ακόμα (δηλαδή δεν υπάρχει ακόμα στο `sys.modules`), αυτό αρχικοποιεί το module· διαφορετικά απλώς επιστρέφει την τιμή του `sys.modules["<modulename>"]`. Σημειώστε ότι δεν εισάγει το module σε κανένα namespace – διασφαλίζει μόνο ότι έχει αρχικοποιηθεί και ότι είναι αποθηκευμένη στο `sys.modules`.

Μπορείτε στη συνέχεια να αποκτήσετε πρόσβαση στα χαρακτηριστικά του module (δηλαδή οποιοδήποτε όνομα ορίζεται στο module) ως εξής:

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

Η κλήση `PyObject_SetAttrString()` για αντιστοίχιση σε μεταβλητές στο module λειτουργεί επίσης.

## 5.11 Πως διασυνδέομαι με αντικείμενα C++ από την Python;

Ανάλογα με τις απαιτήσεις σας, υπάρχουν πολλές προσεγγίσεις. Για να το κάνετε αυτό χειροκίνητα, ξεκινήστε διαβάζοντας το the «Extending and Embedding» document . Συνειδητοποιήστε ότι για το σύστημα χρόνου εκτελεστή Python, δεν υπάρχει μεγάλη διαφορά μεταξύ C και C++ – επομένως η στρατηγική της δημιουργίας ενός νέου τύπου Python γύρω από έναν τύπο δομής C (δείκτη) θα λειτουργήσει επίσης για αντικείμενα C++.

Για βιβλιοθήκες C++, δείτε *Το να γράφει C κάποιος είναι δύσκολο· υπάρχουν άλλες εναλλακτικές*.

## 5.12 Πρόσθεσα ένα module χρησιμοποιώντας το αρχείο Setup και το make αποτυγχάνει· γιατί;

Το setup πρέπει να τελειώνει σε μια νέα γραμμή, αν δεν υπάρχει νέα γραμμή, η διαδικασία build αποτυγχάνει. (Για να διορθωθεί αυτό απαιτεί κάποιο κακόβουλο script shell, και αυτό το σφάλμα είναι τόσο μικρό που δεν φαίνεται να αξίζει τον κόπο.)

## 5.13 Πως κάνω debug μια επέκταση;

Όταν χρησιμοποιείτε το GDB με δυναμικά φορτωμένες επεκτάσεις, δεν μπορείτε να ορίσετε σημείο διακοπής στην επέκταση σας μέχρι να φορτωθεί η επέκτασή σας.

Στο αρχείο σας `.gdbinit` (ή διαδραστικά), προσθέστε την εντολή:

```
br _PyImport_LoadDynamicModule
```

Στη συνέχεια, όταν εκτελείτε το GDB:

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

## 5.14 Θέλω να κάνω compile ένα Python module στο σύστημα Linux μου, αλλά λείπουν ορισμένα αρχεία. Γιατί;

Most packaged versions of Python don't include the `/usr/lib/python2.x/config/` directory, which contains various files required for compiling Python extensions.

For Red Hat, install the `python-devel` RPM to get the necessary files.

For Debian, run `apt-get install python-dev`.

## 5.15 Πώς μπορώ να ξεχωρίσω την «ελλιπή εισαγωγή» από την «έγκυρη εισαγωγή»;

Μερικές φορές θέλετε να μιμηθείτε τη συμπεριφορά του διαδραστικού διερμηνέα Python, όπου σας δίνει ένα συνεχόμενο prompt όταν η είσοδος είναι ελλιπής (π.χ. πληκτρολογήσατε την αρχή μιας δήλωσης `if` ή δεν κλείσατε τις παρενθέσεις ή τα τριπλά εισαγωγικά συμβολοσειρών), αλλά σας δίνει ένα μήνυμα συντακτικού σφάλματος αμέσως όταν η εισαγωγή δεν είναι έγκυρη.

Στην Python μπορείτε να χρησιμοποιήσετε το `module codeop`, η οποία προσεγγίζει επαρκώς τη συμπεριφορά του `parser`. Το `IDLE` χρησιμοποιεί αυτό, για παράδειγμα.

Ο ευκολότερος τρόπος για να το κάνετε στη C είναι να καλέσετε τη `PyRun_InteractiveLoop()` (ίσως σε ξεχωριστό νήμα (thread)) και να αφήσετε τον διερμηνέα Python να χειριστεί την είσοδο για εσάς. Μπορείτε επίσης να ορίσετε `PyOS_ReadlineFunctionPointer()` για να δείξετε την δικιάς προσαρμοσμένη συνάρτηση εισαγωγής. Δείτε τα `Modules/readline.c` και `Parser/myreadline.c` για περισσότερες συμβουλές.

## 5.16 Πώς μπορώ να βρω απροσδιόριστα σύμβολα g++ `__builtin_new` ή `__pure_virtual`;

Για δυναμική φόρτωση module επέκτασης g++, πρέπει να κάνετε recompile την Python, να τη συνδέσετε ξανά χρησιμοποιώντας g++ (αλλάξτε το LINKCC στο Python Modules Makefile), και να συνδέσετε το module επέκτασης σας χρησιμοποιώντας g++ (π.χ. `g++ -shared -o mymodule.so mymodule.o`).

## 5.17 Μπορώ να δημιουργήσω μια κλάση αντικειμένου με ορισμένες μεθόδους που υλοποιούνται στη C και άλλες στη Python (π.χ. μέσω κληρονομικότητας);

Ναι, μπορείτε να κληρονομήσετε από ενσωματωμένες (built-in) κλάσεις όπως `int`, `list`, `dict`, κ.λπ.

Η βιβλιοθήκη Boost Python Library (BPL, <https://www.boost.org/libs/python/doc/index.html>) παρέχει ένα τρόπο για να γίνει αυτό από την C++ (δηλαδή μπορείτε να κληρονομήσετε από μια κλάση επέκτασης γραμμένη σε C++ χρησιμοποιώντας το BPL).

## Python στα Windows FAQ

## 6.1 Πως μπορώ να εκτελέσω ένα πρόγραμμα Python στα Windows;

Αυτό δεν είναι απαραίτητα μια απλή ερώτηση. Εάν είστε ήδη εξοικειωμένοι με την εκτέλεση προγραμμάτων από τη γραμμή εντολών των Windows τότε όλα θα φαίνονται προφανή” διαφορετικά, μπορεί να χρειαστεί λίγη περισσότερη καθοδήγηση.

Εκτός και αν χρησιμοποιείτε κάποιο είδος ενσωματωμένου περιβάλλοντος ανάπτυξης, θα καταλήξετε πληκτρολογώντας εντολές των Windows σε αυτό που αναφέρεται ως «Command prompt window». Συνήθως μπορείτε να δημιουργήσετε ένα τέτοιο παράθυρο από τη γραμμή αναζήτησης σας κάνοντας αναζήτηση για cmd. Θα πρέπει να μπορείτε να αναγνωρίσετε πότε έχετε ξεκινήσει ένα τέτοιο παράθυρο επειδή θα δείτε ένα «command prompt» των Windows, η οποία συνήθως μοιάζει με αυτό:

```
C:\>
```

Το γράμμα μπορεί να είναι διαφορετικό, και μπορεί να υπάρχουνε άλλα πράγματα μετά από αυτό, έτσι μπορείτε να δείτε εξίσου εύκολα κάτι σαν:

```
D:\YourName\Projects\Python>
```

ανάλογα με το πως έχει ρυθμιστεί ο υπολογιστής σας και τι άλλο έχετε κάνει με αυτόν. Μόλις ξεκινήσετε ένα τέτοιο παράθυρο, είστε σε καλό δρόμο για την εκτέλεση προγραμμάτων Python.

Πρέπει να συνειδητοποιήσετε ότι τα Python σενάρια σας πρέπει να επεξεργαστούν από άλλο πρόγραμμα που ονομάζεται *Python interpreter*. Ο interpreter διαβάζει το σενάριο σας, το κάνει compile σε bytecodes και στη συνέχεια εκτελεί τα bytecodes για να τρέξει το πρόγραμμά σας. Λοιπόν, πως ρυθμίζετε τον interpreter για να χειριστεί την Python σας;

Πρώτα από όλα, πρέπει να σιγουρευτείτε ότι το παράθυρο εντολών σας αναγνωρίζει την λέξη «py» ως οδηγία για την έναρξη του interpreter. Εάν έχετε ανοίξει ένα παράθυρο εντολών, θα πρέπει να δοκιμάσετε να εισαγάγετε την εντολή `py` και να πατήσετε `return`:

```
C:\Users\YourName> py
```

Στη συνέχεια, θα πρέπει να δείτε κάτι σαν:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)] on_
↳win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Έχετε ξεκινήσει τον interpreter σε «interactive mode». Αυτό σημαίνει ότι μπορείτε να εισάγετε δηλώσεις ή εκφράσεις Python διαδραστικά και να τις εκτελείτε ή να τις αξιολογείτε ενώ περιμένετε. Αυτό είναι ένα από τα ισχυρότερα χαρακτηριστικά της Python. Ελέγξτε το εισάγοντας μερικές εκφράσεις της επιλογής σας και βλέποντας τα αποτελέσματα:

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

Πολλοί άνθρωποι χρησιμοποιούν τη διαδραστική λειτουργία ως μια βολική αλλά εξαιρετικά προγραμματιζόμενη αριθμομηχανή. Όταν θέλετε να τερματίσετε τη διαδραστική συνεδρία Python, καλέστε τη συνάρτηση `exit()` ή κρατήστε πατημένο το πλήκτρο `Ctrl` ενώ εισάγετε ένα `Z`, και μετά πατήστε το πλήκτρο «Enter» για να επιστρέψετε στο command prompt των Windows.

Μπορεί επίσης να διαπιστώσετε ότι έχετε μια καταχώρηση στο Start-menu όπως *Start ▶ Programs ▶ Python 3.x ▶ Python (command line)* που έχει ως αποτέλεσμα να βλέπετε το `>>>` σε ένα νέο παράθυρο, το παράθυρο θα εξαφανιστεί αφού καλέσετε τη συνάρτηση `exit()` ή εισαγάγετε τον χαρακτήρα `Ctrl-Z` τα Windows εκτελούν μία μόνο εντολή «python» στο παράθυρο και την κλείνουν όταν τερματίζετε τον interpreter.

Τώρα που γνωρίζουμε ότι η εντολή `py` αναγνωρίζεται, μπορείτε να της δώσετε το σενάριο Python σας. Θα πρέπει να δώσετε είτε μια απόλυτη είτε μια σχετική διαδρομή προς στο σενάριο Python. Ας υποθέσουμε ότι το σενάριο Python σας βρίσκεται στην επιφάνεια εργασίας σας και ονομάζεται `hello.py`, και η γραμμή εντολών σας ανοίγει όμορφα στον αρχικό σας κατάλογο, ώστε να βλέπετε κάτι παρόμοιο με:

```
C:\Users\YourName>
```

Λοιπόν τώρα θα ζητήσετε από την εντολή `py` να δώσει το σενάριο σας στην Python πληκτρολογώντας `py` ακολουθούμενη από τη διαδρομή του σεναρίου σας:

```
C:\Users\YourName> py Desktop\hello.py
hello
```

## 6.2 Πως κάνω τα Python scripts εκτελέσιμα;

Στα Windows, το τυπικό πρόγραμμα εγκατάστασης Python ήδη συσχετίζει την επέκταση `.py` με έναν τύπο αρχείου μια ανοιχτή εντολή που εκτελεί τον διερμηνέα (`D:\Program Files\Python\python.exe %1" %*`). Αυτό είναι αρκετό για να κάνετε τα scripts εκτελέσιμα από τη γραμμή εντολών ως `“foo.py”`. Εάν προτιμάτε να μπορείτε να εκτελέσετε το σενάριο πληκτρολογώντας απλά `“foo”` χωρίς επέκταση, πρέπει να προσθέσετε `.py` στη μεταβλητή περιβάλλοντος `PATHTEXT`.



## 6.3 Γιατί μερικές φορές η Python αργεί τόσο πολύ να ξεκινήσει;

Συνήθως η Python ξεκινά πολύ γρήγορα στα Windows, αλλά περιστασιακά υπάρχουν αναφορές σφαλμάτων ότι η Python αρχίζει ξαφνικά να παίρνει πολύ χρόνο για να ξεκινήσει. Αυτό γίνεται ακόμα πιο αινιγματικό, επειδή η Python θα λειτουργεί καλά σε άλλα συστήματα Windows που φαίνεται να έχουν διαμορφωθεί με τον ίδιο τρόπο.

Το πρόβλημα μπορεί να οφείλεται σε εσφαλμένη διαμόρφωση του λογισμικού ελέγχου ιών στο προβληματικό μηχάνημα. Ορισμένοι σαρωτές ιών είναι γνωστό ότι εισάγουν επιβάρυνση εκκίνησης δύο τάξεων μεγέθους όταν ο σαρωτής έχει ρυθμιστεί να παρακολουθεί όλες τις αναγνώσεις από το σύστημα αρχείων. Δοκιμάστε να ελέγξετε τη διαμόρφωση του λογισμικού ιών στα συστήματά σας για να βεβαιωθείτε ότι έχουν όντως διαμορφωθεί πανομοιότυπα. Το McAfee, όταν έχει ρυθμιστεί να σαρώνει όλη τη δραστηριότητα ανάγνωσης του συστήματος αρχείων, είναι ένας συγκεκριμένος φταίχτης.

## 6.4 Πώς μπορώ να δημιουργήσω ένα εκτελέσιμο από ένα σενάριο Python;

Βλ. *Πως μπορώ να δημιουργήσω ένα stand-alone binary από ένα Python script*; για μια λίστα εργαλείων που μπορούν να χρησιμοποιηθούν για τη δημιουργία εκτελέσιμων αρχείων.

## 6.5 Είναι ένα αρχείο \*.pyd ίδιο με ένα αρχείο DLL;

Ναι, τα αρχεία .pyd είναι dll, αλλά υπάρχουν μερικές διαφορές. Εάν έχετε ένα DLL με το όνομα foo.pyd, τότε πρέπει να έχει μια συνάρτηση PyInit\_foo(). Στη συνέχεια μπορείτε να γράψετε Python «import foo», και η Python θα αναζητήσει το foo.pyd (καθώς και το foo.py, foo.pyc) και να το βρει, θα προσπαθήσει να καλέσει το PyInit\_foo() για να το αρχικοποιήσει. Δεν συνδέετε το .exe σας με το foo.lib, καθώς αυτό θα έκανα τα Windows να απαιτήσουν την παρουσία DLL.

Λάβετε υπόψη ότι η διαδρομή αναζήτησης για το foo.pyd είναι το PYTHONPATH, δεν είναι ίδια με τη διαδρομή που χρησιμοποιούν τα Windows για την αναζήτηση του foo.dll. Επίσης, το foo.pyd δεν χρειάζεται να υπάρχει για την εκτέλεση του προγράμματος σας, ενώ εάν συνδέσετε το πρόγραμμά σας με ένα dll, απαιτείται το dll. Φυσικά, απαιτείται το foo.pyd εάν θέλετε να πείτε import foo. Σε ένα DLL, η σύνδεση δηλώνεται στον πηγαίο κώδικα με το \_\_declspec(dllexport). Σε ένα .pyd, η σύνδεση ορίζεται σε μια λίστα διαθέσιμων συναρτήσεων.

## 6.6 Πώς μπορώ να ενσωματώσω την Python σε μια εφαρμογή Windows;

Η ενσωμάτωση του Python interpreter σε μια Windows εφαρμογή μπορεί να συνοψιστεί ως εξής:

1. Να **μην** δημιουργείτε απευθείας την Python στο αρχείο σας .exe. Στα Windows, η Python πρέπει να είναι DLL για να χειρίζεται την εισαγωγή λειτουργικών μονάδων που είναι τα ίδια DLL. (Αυτό είναι το πρώτο κλειδί που δεν τεκμηριώνεται.) Αντίθετα, συνδέστε το pythonNN.dll είναι συνήθως εγκατεστημένο στο C:\Windows\System. NN είναι η έκδοση Python, ένας αριθμός όπως «33» για Python 3.3.

Μπορείτε να συνδεθείτε με την Python με δύο διαφορετικούς τρόπους. Σύνδεση χρόνου φόρτωσης σημαίνει σύνδεση ενάντια στο pythonNN.lib, ενώ η σύνδεση χρόνου εκτέλεσης σημαίνει σύνδεση ενάντια στο pythonNN.dll. (Γενική σημείωση: pythonNN.lib είναι το λεγόμενο «import lib» που αντιστοιχεί στο pythonNN.dll. Απλώς ορίζει σύμβολα για το σύνδεσμο.)

Η σύνδεση χρόνου εκτέλεσης απλοποιεί σημαντικά τις επιλογές συνδέσμων. Όλα συμβαίνουν κατά την εκτέλεση. Ο κώδικας σας πρέπει να φορτώσει `pythonNN.dll` χρησιμοποιώντας την ρουτίνα `LoadLibraryEx()` των Windows. Ο κώδικας πρέπει επίσης να χρησιμοποιεί ρουτίνες πρόσβασης και δεδομένα στο `pythonNN.dll` (δηλαδή, το C API της Python) χρησιμοποιώντας δείκτες που λαμβάνονται από τη ρουτίνα των Windows `GetProcAddress()`. Οι μακροεντολές μπορούν να κάνουν χρησιμοποιώντας αυτούς τους δείκτες διαφανείς σε οποιονδήποτε κώδικα C που καλεί ρουτίνες στο C API της Python.

- Εάν χρησιμοποιείτε SWIG, είναι εύκολο να δημιουργήσετε ένα Python «extension module» που θα κάνει τα δεδομένα και τις μεθόδους της εφαρμογής διαθέσιμα στην Python. Το SWIG θα χειριστεί σχεδόν όλες τις *grungy* λεπτομέρειες για εσάς. Το αποτέλεσμα είναι ο κώδικας C που συνδέσετε στο αρχείο σας .exe (!) Δεν χρειάζεται να δημιουργήσετε ένα αρχείο DLL και αυτό απλοποιεί επίσης τη σύνδεση.
- Το SWIG θα δημιουργήσει μια συνάρτηση `init` (μια συνάρτηση C) της οποίας το όνομα εξαρτάται από το όνομα της μονάδας επέκτασης. Για παράδειγμα, εάν το όνομά του module είναι `leo`, η συνάρτησης `init` θα ονομάζεται `initleo()`. Εάν χρησιμοποιείτε τη σκιά SWIG κλάσεις, όπως θα έπρεπε, η συνάρτησης `init` θα ονομάζεται `initleo()`. Αυτό εκκινεί μια κυρίως κρυφή βοηθητική κλάση που χρησιμοποιείται από την κλάση σκιά.

Ο λόγος για τον οποίο μπορείτε να συνδέσετε τον κώδικα C στο βήμα 2 στο αρχείο σας .exe είναι ότι η κλήση της συνάρτησης αρχικοποίησης ισοδυναμεί με την εισαγωγή του module στην Python! (Αυτό είναι το δεύτερο κλειδί που δεν τεκμηριώνεται.)

- Με λίγα λόγια, μπορείτε να χρησιμοποιήσετε τον ακόλουθο κώδικα για να αρχικοποιήσετε τον Python interpreter με το module επέκτασης.

```
#include <Python.h>
...
Py_Initialize(); // Initialize Python.
initmyAppc(); // Initialize (import) the helper class.
PyRun_SimpleString("import myApp"); // Import the shadow class.
```

- Υπάρχουν δύο προβλήματα με το C API της Python, τα οποία θα γίνουν εμφανή εάν χρησιμοποιήσετε έναν compiler διαφορετικό από τον MSVC, τον compiler που χρησιμοποιήθηκε για τη δημιουργία του `pythonNN.dll`.

Πρόβλημα 1: Οι λεγόμενες συναρτήσεις «Πολύ Υψηλού Επιπέδου» που λαμβάνουν ορίσματα `FILE *` δεν θα λειτουργούν σε multi-compiler περιβάλλον, επειδή η έννοια κάθε μεταγλωττιστή για ένα struct `FILE` θα είναι διαφορετική. Από την άποψη της εφαρμογής, πρόκειται για λειτουργίες πολύ χαμηλού επιπέδου.

Πρόβλημα 2: Το SWIG δημιουργεί τον ακόλουθο κώδικα όταν δημιουργεί wrappers σε void συναρτήσεις:

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

Δυστυχώς, το `Py_None` είναι μια μακροεντολή που επεκτείνεται σε μια αναφορά σε μια σύνθετη δομή δεδομένων που ονομάζεται `_Py_NoneStruct` μέσα στο `pythonNN.dll`. Και πάλι, αυτός ο κώδικας θα αποτύχει σε ένα multi-compiler περιβάλλον. Αντικαταστήστε αυτόν τον κώδικα με:

```
return Py_BuildValue("");
```

Μπορεί να είναι δυνατή η χρήση της εντολής `%typemap` της SWIG για να γίνει αυτόματα η αλλαγή, αν και δεν κατάφερα να το κάνω αυτό να λειτουργήσει (είμαι εντελώς αρχάριος της SWIG).

- Η χρήση ενός Python shell script για να δημιουργήσετε ένα Python interpreter παράθυρο μέσα από την εφαρμογή σας Windows δεν είναι καλή ιδέα” το παράθυρο που θα προκύψει θα είναι ανεξάρτητο από το σύστημα παραθύρου της εφαρμογής σας. Αντίθετα, εσείς (ή η κλάση `wxPythonWindow`) θα πρέπει να δημιουργήσετε ένα «εγγενές» (native) interpreter παράθυρο. Είναι εύκολο να συνδέσετε αυτό το παράθυρο με

τον Python interpreter. Μπορείτε να ανακατευθύνετε το i/o της Python στο `_any_` αντικείμενο που υποστηρίζει ανάγνωση και εγγραφή, επομένως το μόνο που χρειάζεστε είναι ένα αντικείμενο Python (που ορίζεται στο module επέκτασης) που περιέχει τις μεθόδους `read()` και `write()`.

## 6.7 Πως μπορώ να εμποδίσω τους editors να εισάγουν tabs στον πηγαίο της Python μου;

Το FAQ δεν συνιστούν την χρήση καρτελών και ο οδηγός στυλ Python, [PEP 8](#), συνιστά 4 κενά για τον κατανα-  
μημένο κώδικα Python” αυτή είναι επίσης η προεπιλεγμένη λειτουργία python του Emacs.

Σε οποιονδήποτε editor, η μίξη tabs και spaces είναι κακή ιδέα. Το MSVC δεν διαφέρει από αυτή την άποψη και ρυθμίζεται εύκολα για να χρησιμοποιεί κενά: Πάρτε *Tools* ▶ *Options* ▶ *Tabs*, και για τον τύπο αρχείου «Default» ορίστε το «Tab size» και το «Indent size» στο 4, και επιλέξτε το κουμπί επιλογής «Insert spaces».

Η Python εγείρει το `IndentationError` ή `TabError` εάν τα μικτά tabs και spaces προκαλούν προβλήματα στο `leading whitespace`. Μπορείτε επίσης να εκτελέσετε το module `tabnanny` για να ελέγξετε εάν δέντρο καταλόγου σε λειτουργία batch.

## 6.8 Πώς μπορώ να ελέγξω για ένα πάτημα πλήκτρων χωρίς αποκλεισμό;

Χρησιμοποιείτε το module `msvcrt`. Αυτή είναι ένα τυπικό module επέκτασης ειδικά για τα Windows. Ορίζει μια συνάρτηση `kbhit()` η οποία ελέγχει εάν υπάρχει ένα χτύπημα πληκτρολογίου, και το `getch()` παίρνει έναν χαρακτήρα χωρίς να τον επαναλαμβάνει.

## 6.9 Πως μπορώ να διορθώσω το σφάλμα που λείπει το `api-ms-win-crt-runtime-l1-1-0.dll`;

Αυτό μπορεί να συμβεί σε Python 3.5 και νεότερες εκδόσεις όταν χρησιμοποιείτε Windows 8.1 ή νεότερη έκδοση χωρίς να έχουν εγκατασταθεί όλες οι ενημερώσεις. Πρώτα βεβαιωθείτε ότι το λειτουργικό σας σύστημα υποστη-  
ρίζεται και είναι ενημερωμένο και εάν αυτό δεν επιλύσει το πρόβλημα, επισκεφθείτε τη [σελίδα υποστήριξης της Microsoft](#) για καθοδήγηση σχετικά με τη μη αυτόματη εγκατάσταση της ενημέρωσης C Runtime.



## 7.1 General GUI Questions

## 7.2 What GUI toolkits exist for Python?

Standard builds of Python include an object-oriented interface to the Tcl/Tk widget set, called tkinter. This is probably the easiest to install (since it comes included with most [binary distributions](#) of Python) and use. For more info about Tk, including pointers to the source, see the [Tcl/Tk home page](#). Tcl/Tk is fully portable to the macOS, Windows, and Unix platforms.

Depending on what platform(s) you are aiming at, there are also several alternatives. A [list of cross-platform](#) and [platform-specific](#) GUI frameworks can be found on the python wiki.

## 7.3 Tkinter questions

### 7.3.1 How do I freeze Tkinter applications?

Freeze is a tool to create stand-alone applications. When freezing Tkinter applications, the applications will not be truly stand-alone, as the application will still need the Tcl and Tk libraries.

One solution is to ship the application with the Tcl and Tk libraries, and point to them at run-time using the `TCL_LIBRARY` and `TK_LIBRARY` environment variables.

To get truly stand-alone applications, the Tcl scripts that form the library have to be integrated into the application as well. One tool supporting that is SAM (stand-alone modules), which is part of the Tix distribution (<https://tix.sourceforge.net/>).

Build Tix with SAM enabled, perform the appropriate call to `Tclsam_init()`, etc. inside Python's `Modules/tkappinit.c`, and link with `libtclsam` and `libtkjam` (you might include the Tix libraries as well).

### 7.3.2 Can I have Tk events handled while waiting for I/O?

On platforms other than Windows, yes, and you don't even need threads! But you'll have to restructure your I/O code a bit. Tk has the equivalent of X's `XtAddInput()` call, which allows you to register a callback function which will be called from the Tk mainloop when I/O is possible on a file descriptor. See `tkinter-file-handlers`.

### 7.3.3 I can't get key bindings to work in Tkinter: why?

An often-heard complaint is that event handlers bound to events with the `bind()` method don't get handled even when the appropriate key is pressed.

The most common cause is that the widget to which the binding applies doesn't have «keyboard focus». Check out the Tk documentation for the `focus` command. Usually a widget is given the keyboard focus by clicking in it (but not for labels; see the `takefocus` option).

---

## «Γιατί είναι εγκατεστημένη η Python στον υπολογιστή μου;» FAQ

---

### 8.1 Τι είναι η Python;

Η Python είναι μια γλώσσα προγραμματισμού. Χρησιμοποιείται για πολλές διαφορετικές εφαρμογές. Χρησιμοποιείται σε ορισμένα λύκεια και κολέγια ως εισαγωγική γλώσσα προγραμματισμού επειδή η Python είναι εύκολη στην εκμάθηση, αλλά χρησιμοποιείται επίσης από επαγγελματίες προγραμματιστές λογισμικού σε μέρη όπως στην Google, στην NASA και στην Lucasfilm Ltd.

Αν θέλετε να μάθετε περισσότερα για την Python, ξεκινήστε με τον [Οδηγό για αρχάριους στην Python](#).

### 8.2 Γιατί είναι εγκατεστημένη η Python στον υπολογιστή μου;

Αν βρείτε την Python εγκατεστημένη στο σύστημά σας, αλλά δεν θυμάστε να την έχετε εγκαταστήσει υπάρχουν διάφοροι πιθανοί τρόποι με τους οποίους θα μπορούσε να έχει φτάσει εκεί.

- Ίσως κάποιος άλλος χρήστης του υπολογιστή ήθελε να μάθει προγραμματισμό και το εγκατέστησε. Θα πρέπει να βρείτε ποιος χρησιμοποιούσε τον υπολογιστή και μπορεί να το εγκατέστησε.
- Μια εφαρμογή τρίτου μέρους που έχει εγκατασταθεί στον υπολογιστή μπορεί να έχει γραφτεί σε Python και να περιελάμβανε μια εγκατάσταση της Python. Υπάρχουν πολλές τέτοιες εφαρμογές, από προγράμματα GUI μέχρι διακομιστές δικτύου και διαχειριστικά scripts.
- Ορισμένες συσκευές με Windows έχουν επίσης εγκατεστημένη την Python. Αυτή τη στιγμή γνωρίζουμε υπολογιστές της Hewlett-Packard και της Compaq που περιλαμβάνουν την Python. Προφανώς ορισμένα από τα εργαλεία διαχείρισης της HP/Compaq είναι γραμμένα σε Python.
- Πολλά λειτουργικά συστήματα συμβατά με Unix, όπως το macOS και ορισμένες διανομές Linux, έχουν από προεπιλογή εγκατεστημένη την Python, η οποία περιλαμβάνεται στη βασική εγκατάσταση.

## 8.3 Μπορώ να διαγράψω την Python;

Αυτό εξαρτάται από το πού προήλθε η Python.

Αν κάποιος το εγκατέστησε σκόπιμα, μπορείτε να το αφαιρέσετε χωρίς να βλάψετε τίποτα. Στα Windows, χρησιμοποιήστε το εικονίδιο Προσθήκη/Αφαίρεση προγραμμάτων στον Πίνακα Ελέγχου.

Αν η Python εγκαταστάθηκε από μια εφαρμογή τρίτου μέρους, μπορείτε επίσης να την αφαιρέσετε, αλλά η εφαρμογή αυτή δεν θα λειτουργεί πλέον. Θα πρέπει να χρησιμοποιήσετε το πρόγραμμα απεγκατάστασης αυτής της εφαρμογής αντί να αφαιρέσετε απευθείας την Python.

Εάν η Python περιλαμβάνεται στο λειτουργικό σας σύστημα, δεν συνιστάται η αφαίρεσή της. Αν την αφαιρέσετε, όποια εργαλεία ήταν γραμμένα σε Python δεν θα λειτουργούν πλέον, και κάποια από αυτά μπορεί να είναι σημαντικά για εσάς. Στη συνέχεια, θα απαιτηθεί επανεγκατάσταση ολόκληρου του συστήματος για να διορθώσετε ξανά τα πράγματα.



>>>

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

...

Μπορεί να αναφέρεται σε:

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

### 2to3

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

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

### αφηρημένη βασική κλάση

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

### annotation

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

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

Βλ. [variable annotation](#), [function annotation](#), [PEP 484](#) και [PEP 526](#), τα οποία περιγράφουν την λειτουργικότητά. Επίσης βλ. [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 αρχής μιας λίστας ορισμάτων ή/και να μεταβιβάζονται ως στοιχεία ενός *iterable* πριν από `*`. Για παράδειγμα, το 3 και το 5 αποτελούν ορίσματα θέσης στις παρακάτω κλήσεις:

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

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

Βλ. επίσης την εγγραφή του 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

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

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.

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

**ασύγχρονος iterable**

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**.

**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 `identifiers`, 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 διαβάζει 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 δυναδικών δεδομένα σε αμετάβλητα αντικείμενα («δυναδικά 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).

**μιγαδικός αριθμός**

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

**διαχειριστής context**

Ένα αντικείμενο που ελέγχει το περιβάλλον που εμφανίζεται σε μια δήλωση `with` ορίζοντας τις μεθόδους `__enter__()` και `__exit__()`. Βλ. **PEP 343**.

**context μεταβλητή**

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

**contiguous**

Ένα buffer θεωρείται contiguous ακριβώς εάν είναι είτε *C-contiguous* είτε *Fortran 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](#). Ο όρος «CPython» χρησιμοποιείται όταν είναι απαραίτητο για την διάκριση αυτής της υλοποίησης από άλλες όπως η *Jython* ή η *IronPython*.

**decorator**

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

Η σύνταξη του decorator είναι απλώς καλλωπιστική, οι ακόλουθοι δύο ορισμοί συναρτήσεων είναι σημειολογικά 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 descriptors` ή το Πρακτικός οδηγός για τη χρήση του Descriptor.

**λεξικό**

Ένα προσαρτησιακός πίνακας, 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. `comprehensions`.

**όψη λεξικού**

Τα αντικείμενα που επιστρέφονται από `dict.keys()`, `dict.values()`, και `dict.items()` καλούνται όψεις λεξικού. Αυτές παρέχουν μια δυναμική όψη των των εγγραφών του λεξικού, που σημαίνει ότι

όταν το λεξικό μεταβάλλεται, η όψη αντικατοπτρίζει αυτές τις αλλαγές. Για να αναγκάσετε την όψη λεξικού να γίνει μια πλήρης λίστα χρησιμοποιήστε το `list(dictview)`. Βλ. [dict-views](#).

### docstring

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

### duck-typing

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

### EAFP

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

### έκφραση

Ένα κομμάτι σύνταξης που μπορεί να αξιολογηθεί σε κάποια τιμή. Με άλλα λόγια, μια έκφραση είναι μια συσσώρευση στοιχείων έκφρασης όπως κυριολεξία, ονόματα, πρόσβαση χαρακτηριστικών, τελεστές ή κλήσεις συναρτήσεων που όλες επιστρέφουν μια τιμή. Σε αντίθεση με πολλές άλλες γλώσσες, δεν είναι όλες οι γλωσσικές δομές εκφράσεις. 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

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

### αντικείμενο αρχείου

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 [δυναμικά αρχεία](#), buffered [δυναμικά 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 εγγυηθεί 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.

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

See [PEP 302](#), [PEP 420](#) and [PEP 451](#) for much more detail.

### ακέραια διαίρεση

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

### συνάρτηση

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

### 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*.

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 πότε μια νέα δυνατότητα προστέθηκε για πρώτη φορά στην γλώσσα και πότε θα γίνει (ή έγινε) η προεπιλογή:

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

### συλλογή απορριμάτων

Η διαδικασία απελευθέρωσης της μνήμης όταν δεν χρησιμοποιείται άλλο. Η Python εκτελεί συλλογή απορριμάτων μέσω καταμέτρησης αναφορών και ενός κυκλικού συλλέκτη σκουπιδιών που είναι σε θέση να

ανιχνεύει και να σπάει τους κύκλους αναφοράς. Ο συλλέκτης 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*.

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

### generator έκφραση

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

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

### γενική συνάρτηση

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

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

### γενικός τύπος

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

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

### GIL

Βλ. *global interpreter lock*.

### global interpreter lock

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

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.

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



**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. `pyc-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 ισοτιμία 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.

**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 τις άμεσα και εμφανίζοντας τα αντικείμενα. Απλώς εκκινήστε την `python` χωρίς ορίσματα (πιθανώς επιλέγοντας το από το κύριο μενού του υπολογιστή σας). Αποτελεί έναν αποδοτικό τρόπο για να δοκιμάστε νέες ιδέες ή να εξετάσετε λειτουργικές μονάδες και πακέτα (θυμηθείτε `help(x)`).

**interpreted**

Η Python είναι μια *interpreted* γλώσσα, σε αντίθεση με μια μεταγλωττισμένη, αν και η διάκριση μπορεί να είναι και θολή λόγω της παρουσίας 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 απε-

λευθερώνει 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 διεργασίας είναι ότι το `__main__` module ή ολοκληρώθηκε η εκτέλεση του κώδικα που έτρεχε.

### iterable

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

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`.

**Λεπτομέρεια υλοποίησης CPython:** Το CPython δεν εφαρμόζει με συνέπεια την απαίτηση να ορίζει ένας iterator `__iter__()`.

### 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.

### 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

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

### 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**

Method Resolution Order is the order in which base classes are searched for a member during lookup. See [The Python 2.3 Method Resolution Order](#) for details of the algorithm used by the Python interpreter since the 2.3 release.

**module**

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

Βλ. επίσης *package*.

**τεχνικές προδιαγραφές module**

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

**MRO**

Βλ. *method resolution order*.

**mutable**

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

**named tuple**

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

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

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

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

A **PEP 420** *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

Βλ. επίσης *module*.

### nested scope

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

### κλάση νέου στυλ

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*.

### 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 *η διαφορά μεταξύ ορισμάτων και παραμέτρων*, την κλάση `inspect.Parameter`, την ενότητα *function* 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 συμβάλουν 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__
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
'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

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

### κανονικό πακέτο

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

Βλ. επίσης *namespace package*.

### \_\_slots\_\_

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

### ακολουθία

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

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. comprehensions.

### μοναδικό dispatch

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

### slice

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

### ειδική μέθοδος

Μια μέθοδος που καλείται σιωπηρά από την Python για να εκτελέσει μια συγκεκριμένη λειτουργία σε έναν



τύπο, όπως η προσθήκη. Τέτοιες μέθοδοι έχουν ονόματα που ξεκινούν και τελειώνουν με διπλές κάτω παύλες. Οι ειδικές μέθοδοι τεκμηριώνονται στο `specialnames`.

### 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`.

### 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*.

### κωδικοποίηση κειμένου

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

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

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

### αρχείο κειμένου

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

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

### συμβολοσειρά τριπλών εισαγωγικών

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

### τύπος

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

### type alias

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

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

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

μπορεί να γίνει πιο ευανάγνωστο όπως:

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

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

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

### type hint

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

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 `annassign`.

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`.

### Zen της Python

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

## ΠΑΡΑΡΤΗΜΑ Β΄

---

### About these documents

---

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

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

Πολλές ευχαριστίες πηγαίνουν στους:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- 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.

### B'.1 Contributors to the Python Documentation

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

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



## Γ'.1 Η ιστορία του λογισμικού

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

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

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

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

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

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

---

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

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

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

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

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

### Γ'.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.11.14

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),  
→and  
the Individual or Organization ("Licensee") accessing and otherwise using  
→Python  
3.11.14 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby  
grants Licensee a nonexclusive, royalty-free, world-wide license to  
→reproduce,  
analyze, test, perform and/or display publicly, prepare derivative works,  
distribute, and otherwise use Python 3.11.14 alone or in any derivative  
version, provided, however, that PSF's License Agreement and PSF's notice  
→of  
copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All  
→Rights  
Reserved" are retained in Python 3.11.14 alone or in any derivative version  
prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or  
incorporates Python 3.11.14 or any part thereof, and wants to make the  
derivative work available to others as provided herein, then Licensee  
→hereby  
agrees to include in any such work a brief summary of the changes made to  
→Python  
3.11.14.
4. PSF is making Python 3.11.14 available to Licensee on an "AS IS" basis.  
PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF

- EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION  
→OR  
WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT  
→THE  
USE OF PYTHON 3.11.14 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.11.14  
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT  
→OF  
MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.11.14, OR ANY  
→DERIVATIVE  
THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach  
→of  
its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any  
→relationship  
of agency, partnership, or joint venture between PSF and Licensee. This  
→License  
Agreement does not grant permission to use PSF trademarks or trade name in  
→a  
trademark sense to endorse or promote products or services of Licensee, or  
→any  
third party.
8. By copying, installing or otherwise using Python 3.11.14, Licensee agrees  
to be bound by the terms and conditions of this License Agreement.

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

### ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ ΑΝΟΙΧΤΟΥ ΚΩΔΙΚΑ BEOPEN PYTHON ΕΚΔΟΣΗ 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## Γ'.2.3 ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ CNRI ΓΙΑ PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the internet using the following URL: <http://hdl.handle.net/1895.22/1013>."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR

(συνέχεια στην επόμενη σελίδα)



(συνεχίζεται από την προηγούμενη σελίδα)

<p>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.</p> <p>6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.</p> <p>7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.</p> <p>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.</p>
--

## Γ'.2.4 ΣΥΜΦΩΝΙΑ ΑΔΕΙΑΣ CWI ΓΙΑ PYTHON 0.9.0 ΕΩΣ 1.2

<p>Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.</p> <p>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.</p> <p>STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.</p>
--

## Γ'.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.11.14 DOCUMENTATION

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

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

## Γ'.3 Άδειες και Ευχαριστίες για Ενσωματωμένο Λογισμικό

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

### Γ'.3.1 Mersenne Twister

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

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

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

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

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

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

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

### Γ'.3.2 Sockets

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

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

```
Copyright 1996 by Sam Rushing
```

```
    All Rights Reserved
```

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

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

### Γ'.3.4 Διαχείριση Cookie

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

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

```
    All Rights Reserved
```

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

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

### Γ'.3.5 Ανίχνευση εκτέλεσης

Η ενότητα `trace` περιέχει την παρακάτω ειδοποίηση:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

### Γ'.3.6 Συναρτήσεις `UUencode` και `UUdecode`

Η ενότητα `uu` 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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

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

## Γ'.3.7 Κλήσεις Απομακρυσμένης Διαδικασίας XML

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

```
The XML-RPC client interface is  
  
Copyright (c) 1999-2002 by Secret Labs AB  
Copyright (c) 1999-2002 by Fredrik Lundh  
  
By obtaining, using, and/or copying this software and/or its  
associated documentation, you agree that you have read, understood,  
and will comply with the following terms and conditions:  
  
Permission to use, copy, modify, and distribute this software and  
its associated documentation for any purpose and without fee is  
hereby granted, provided that the above copyright notice appears in  
all copies, and that both that copyright notice and this permission  
notice appear in supporting documentation, and that the name of  
Secret Labs AB or the author not be used in advertising or publicity  
pertaining to distribution of the software without specific, written  
prior permission.  
  
SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD  
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-  
ABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR  
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY  
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,  
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS  
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE  
OF THIS SOFTWARE.
```

## Γ'.3.8 test\_epoll

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
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

### Γ'.3.9 Επιλογή kqueue

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

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.
```

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

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

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

### Γ'.3.10 SipHash24

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

```
<MIT License>
Copyright (c) 2013 Marek Majkowski <marek@popcount.org>
```

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

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

```
</MIT License>
```

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

```
Original location:
  https://github.com/majek/csiphash/

Solution inspired by code from:
  Samuel Neves (supercop/crypto_auth/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`, `ssl`, `crypt` 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 συμπεριλαμβανουμε 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.
```

(συνέχεια στην επόμενη σελίδα)



(συνεχίζεται από την προηγούμενη σελίδα)

"Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

"Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual,

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.
4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
  - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
  - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
  - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
  - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

### 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
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

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

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

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

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

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

### Γ'.3.15 zlib

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

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler
```

```
This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.
```

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:
```

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

```
Jean-loup Gailly
jloup@gzip.org
```

```
Mark Adler
madler@alumni.caltech.edu
```

### Γ'.3.16 cfuhash

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

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

```
This code is released under the BSD license:
```

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

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

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

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

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

### Γ'.3.17 libmpdec

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

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

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

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

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

### Γ'.3.18 W3C C14N σουίτα δοκιμής

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

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.
```

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

(συνέχεια στην επόμενη σελίδα)

(συνεχίζεται από την προηγούμενη σελίδα)

- \* Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.

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

### Γ'.3.19 Audioop

To module audioop χρησιμοποιεί ως βάση κώδικα του αρχείου g771.c του έργου Sox. <https://sourceforge.net/projects/sox/files/sox/12.17.7/sox-12.17.7.tar.gz>

Αυτό ο πηγαίος κώδικας είναι προϊόν της Sun Microsystems, Inc. και παρέχεται για απεριόριστη χρήση. Οι χρήστες μπορούν να αντιγράψουν ή να τροποποιήσουν αυτόν τον πηγαίο κώδικα χωρίς χρέωση.

Ο ΠΗΓΑΙΟΣ ΚΩΔΙΚΑΣ ΤΟΥ SUN ΠΑΡΕΧΕΤΑΙ ΟΠΩΣ ΕΧΕΙ ΧΩΡΙΣ ΚΑΝΕΝΟΣ ΕΙΔΟΥΣ ΕΓΓΥΗΣΕΙΣ ΣΥΜΠΕΡΙΛΑΜΒΑΝΟΜΕΝΩΝ ΕΓΓΥΗΣΕΩΝ ΣΧΕΔΙΑΣΜΟΥ, ΕΜΠΟΡΕΥΣΙΜΟΤΗΤΑΣ ΚΑΙ ΚΑΤΑΛΛΗΛΟΤΗΤΑΣ ΓΙΑ ΣΥΓΚΕΚΡΙΜΕΝΟ ΣΚΟΠΟ Ή ΠΟΥ ΠΡΟΚΥΠΤΕΙ ΑΠΟ ΚΑΠΟΙΑ ΠΟΡΕΙΑ ΣΥΝΑΛΛΑΓΗΣ, ΧΡΗΣΗΣ Ή ΕΜΠΟΡΙΚΗΣ ΠΡΑΚΤΙΚΗΣ.

Ο πηγαίος κώδικας του Sun παρέχεται χωρίς την υποστήριξη και χωρίς καμία υποχρέωση εκ μέρους της Sun Microsystems, Inc. να βοηθήσει στην χρήση, στη διόρθωση, τροποποίηση ή βελτίωση του.

SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS SOFTWARE OR ANY PART THEREOF.

Σε καμία περίπτωση η Sun Microsystems, Inc. δεν φέρει ευθύνη για τυχόν απώλεια εσόδων ή κερδών ή άλλες ειδικές, έμμεσες και επακόλουθες ζημιές, ακόμη και αν η Sun έχει ενημερωθεί για την πιθανότητα τέτοιων ζημιών.

Sun Microsystems, Inc. 2550 Garcia Avenue Mountain View, Καλιφόρνια 94043

### 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
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.
```



## ΠΑΡΑΡΤΗΜΑ Δ'

---

### Copyright

---

Η Python και αυτή η τεκμηρίωση είναι:

Copyright © 2001-2023 Python Software Foundation. Όλα τα δικαιώματα διατηρούνται.

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

- ..., [89](#)
- 2to3, [89](#)
- >>>, [89](#)
- BDFL, [91](#)
- CPython, [93](#)
- C-contiguous, [92](#)
- EAFP, [94](#)
- Fortran contiguous, [92](#)
- GIL, [96](#)
- IDLE, [97](#)
- LBYL, [99](#)
- MRO, [100](#)
- PATH, [62](#)
- PEP, [102](#)
- PYTHONDONTWRITEBYTECODE, [43](#)
- Python 3000, [103](#)
- Python Enhancement Proposals
  - PEP 1, [102](#)
  - PEP 5, [6](#)
  - PEP 8, [11](#), [40](#), [83](#)
  - PEP 238, [95](#)
  - PEP 278, [106](#)
  - PEP 302, [95](#), [99](#)
  - PEP 343, [92](#)
  - PEP 362, [90](#), [102](#)
  - PEP 387, [3](#)
  - PEP 411, [103](#)
  - PEP 420, [95](#), [101](#), [103](#)
  - PEP 443, [96](#)
  - PEP 451, [95](#)
  - PEP 483, [96](#)
  - PEP 484, [90](#), [95](#), [96](#), [106](#)
  - PEP 492, [90](#), [91](#), [93](#)
  - PEP 498, [94](#)
  - PEP 519, [102](#)
  - PEP 525, [90](#)
  - PEP 526, [90](#), [106](#)
  - PEP 572, [49](#)
  - PEP 585, [96](#)
  - PEP 602, [5](#)
  - PEP 3116, [106](#)
  - PEP 3147, [42](#)
  - PEP 3155, [103](#)
- Pythonic, [103](#)
- Zen της Python, [106](#)
- \_\_future\_\_, [95](#)
- \_\_slots\_\_, [104](#)
- annotation, [89](#)
- annotation μεταβλητής, [106](#)
- awaitable, [91](#)
- bytecode, [92](#)
- bytes-like αντικείμενα, [91](#)
- callable, [92](#)
- callback, [92](#)
- context μεταβλητή, [92](#)
- contiguous, [92](#)
- coroutine, [93](#)
- coroutine συνάρτηση, [93](#)
- decorator, [93](#)
- descriptor, [93](#)
- docstring, [94](#)
- duck-typing, [94](#)
- f-string, [94](#)
- finder, [95](#)
- generator, [96](#)
- generator iterator, [96](#)
- generator έκφραση, [96](#)
- global interpreter lock, [96](#)
- hash-based pyc, [97](#)
- hashable, [97](#)
- immutable, [97](#)
- interpreted, [97](#)
- iterable, [98](#)
- iterator, [98](#)
- lambda, [99](#)
- list comprehension, [99](#)
- loader, [99](#)
- magic

- μέθοδος, 99
- mapping, 99
- meta path finder, 99
- module, 100
- module επέκτασης, 94
- mutable, 100
- named tuple, 100
- namespace, 101
- nested scope, 101
- path based finder, 102
- path entry, 102
- path entry finder, 102
- path entry hook, 102
- path-like αντικείμενο, 102
- provisional API, 103
- provisional πακέτο, 103
- set comprehension, 104
- slice, 104
- special
  - μέθοδος, 104
- strong reference, 105
- type alias, 105
- type hint, 106
- virtual environment, 106
- virtual machine, 106

## A

- ακέραια διαίρεση, 95
- ακολουθία, 104
- αναγνωρισμένο όνομα, 103
- αντικείμενο, 101
- αντικείμενο αρχείου, 94
- αντικείμενο που μοιάζει με αρχείο, 94
- αρχείο κειμένου, 105
- ασύγχρονος generator, 90
- ασύγχρονος generator iterator, 90
- ασύγχρονος iterable, 91
- ασύγχρονος iterator, 91
- ασύγχρονος διαχειριστής context, 90
- αφηρημένη βασική κλάση, 89

## Γ

- γενική συνάρτηση, 96
- γενικός τύπος, 96

## Δ

- δανεική αναφορά, 91
- δήλωση, 105
- διαδραστικός, 97
- διαχειριστής context, 92
- δυναμικό αρχείο, 91

## Ε

- ειδική μέθοδος, 104

- εισαγόμενο path, 97
- εισαγωγέας, 97
- εισαγωγή, 97
- έκφραση, 94
- ελεγκτής στατικού τύπου, 105

## K

- καθολικές νέες γραμμές, 106
- κανονικό πακέτο, 104
- κατανόηση λεξικού, 93
- κλάση, 92
- κλάση νέου στυλ, 101
- κωδικοποίηση κειμένου, 105
- κωδικοποίηση συστήματος αρχείων και χειριστής σφαλμάτων, 94

## Λ

- λεξικό, 93
- λίστα, 99

## M

- μαγική μέθοδος, 99
- μέθοδος, 100
  - magic, 99
  - special, 104
- μετα-κλάση, 100
- μεταβλητή κλάσης, 92
- μεταβλητή περιβάλλοντος
  - PATH, 62
  - PYTHONDONTWRITEBYTECODE, 43
- μιγαδικός αριθμός, 92
- μοναδικό dispatch, 104

## O

- όρισμα, 90
  - διαφορά από παράμετρο, 15
- όρισμα keyword, 99
- όρισμα θέσης, 103
- όψη λεξικού, 93

## Π

- πακέτο, 101
- πακέτο namespace, 101
- παράμετρος, 101
  - διαφορά από όρισμα, 15
- πλήθος αναφοράς, 104

## Σ

- σειρά ανάλυσης μεθόδων, 100
- συλλογή απορριμάτων, 95
- συμβολοσειρά τριπλών εισαγωγικών, 105
- συνάρτηση, 95
- συνάρτηση annotation, 95

συνάρτηση key, [98](#)

## T

τερματισμός λειτουργίας διερμηνέα, [97](#)

τεχνικές προδιαγραφές module, [100](#)

τμήμα, [103](#)

τοπική κωδικοποίηση, [99](#)

τύπος, [105](#)

## X

χαρακτηριστικό, [91](#)