The Python Language Reference Release 3.13.0

Guido van Rossum and the Python development team

November 15, 2024

Python Software Foundation Email: docs@python.org

CONTENTS

1	Intro	oduction 3		
	1.1	Alternate	e Implementations	
	1.2	Notation		
2	Lexic	cal analys		
	2.1	Line stru	icture	
		2.1.1	Logical lines	
		2.1.2	Physical lines	
		2.1.3	Comments	
		2.1.4	Encoding declarations	
		2.1.5	Explicit line joining	
			Implicit line joining	
			Blank lines	
			Indentation	
			Whitespace between tokens	
	2.2		kens	
	2.3			
	2.3		rs and keywords	
			Keywords	
			Soft Keywords	
			Reserved classes of identifiers	
	2.4		9	
			String and Bytes literals	
		2.4.2	String literal concatenation	
		2.4.3	f-strings	
		2.4.4	Numeric literals	
			Integer literals	
			Floating-point literals	
			Imaginary literals	
	2.5		rs	
	2.6	Delimite		
	2.0	Demine	13	
3	Data	model	17	
	3.1		values and types	
	3.2		dard type hierarchy	
	3.4		None	
			1	
			Ellipsis	
			numbers.Number	
			Sequences	
			Set types	
			Mappings	
		3.2.8	Callable types 21	
		3.2.9	Modules	
		3.2.10	Custom classes	

		3.2.11 3.2.12	Class instances		29 29
		3.2.13	Internal types	2	9
	3.3	Special	method names		4
		3.3.1	Basic customization		5
		3.3.2	Customizing attribute access		8
		3.3.3	Customizing class creation		2
		3.3.4	Customizing instance and subclass checks		6
		3.3.5	Emulating generic types		6
		3.3.6	Emulating callable objects		8
		3.3.7	Emulating container types		18
		3.3.8	Emulating numeric types		60
		3.3.9	With Statement Context Managers		2
		3.3.10	Customizing positional arguments in class pattern matching		3
		3.3.11	Emulating buffer types		3
		3.3.12	Special method lookup		3
	3.4	Corouti			55
	5.1	3.4.1	Awaitable Objects		55
		3.4.2	Coroutine Objects		55
		3.4.3	Asynchronous Iterators		,5 6
		3.4.4	Asynchronous Context Managers		66
		J.4.4	Asynchronous Context Managers		U
4	Execu	ution mo	odel	5	9
	4.1	Structur	re of a program	5	9
	4.2		g and binding		9
		4.2.1	Binding of names		9
		4.2.2	Resolution of names		60
		4.2.3	Annotation scopes		
		4.2.4	Lazy evaluation		1
		4.2.5	Builtins and restricted execution		2
		4.2.6	Interaction with dynamic features	6	12
	4.3		Interaction with dynamic features		52 52
		Exception	ions	6	52
5	The i	Exception Exception Exception	ystem	6	52 5 5
5	The i 5.1	Exception mport sy import	ystem tlib	6 6	52 55
5	The i	Exception mport sy import	ystem tlibes	66 66	52 55 55
5	The i 5.1	mport sy import Package 5.2.1	ystem tlib	66 66 66	55 55 56
5	The i 5.1	mport sy import Package 5.2.1 5.2.2	ystem tlib es Regular packages Namespace packages	66 66 66	55 55 56
5	The i 5.1	mport sy import Package 5.2.1 5.2.2	ystem tlib	66 66 66 66	55 55 56 56 56
5	The i 5.1 5.2	mport sy import Package 5.2.1 5.2.2	ystem Llib Regular packages Namespace packages The module cache	60	55 55 56 56
5	The i 5.1 5.2	mport sy import Package 5.2.1 5.2.2 Searchin	ystem tlib es Regular packages Namespace packages ng	60	55 55 56 56 56
5	The i 5.1 5.2	import sy import Package 5.2.1 5.2.2 Searchin 5.3.1	ystem Llib Regular packages Namespace packages The module cache	66 66 66 66	55 55 56 56 57 57
5	The i 5.1 5.2	import sy import Package 5.2.1 5.2.2 Searchin 5.3.1 5.3.2	ystem tlib es Regular packages Namespace packages ng The module cache Finders and loaders	66 66 66 66 66	55 55 56 56 57 57
5	The i 5.1 5.2	Exception import sy import sy import Package 5.2.1 5.2.2 Searchin 5.3.1 5.3.2 5.3.3 5.3.4	ystem tlibes Regular packages Namespace packages ng The module cache Finders and loaders Import hooks The meta path	66 66 66 66 66	55 55 56 56 57 57 57
5	The i 5.1 5.2 5.3	mport sy import Package 5.2.1 5.2.2 Searchin 5.3.1 5.3.2 5.3.3 5.3.4 Loading 5.4.1	ystem tlib es Regular packages Namespace packages ng The module cache Finders and loaders Import hooks The meta path g Loaders	66 66 66 66 66 66	55 55 56 56 57 57
5	The i 5.1 5.2 5.3	import sy import sy import Package 5.2.1 5.2.2 Searchin 5.3.1 5.3.2 5.3.3 5.3.4 Loading	ystem tlibes Regular packages Namespace packages ng The module cache Finders and loaders Import hooks The meta path	66 66 66 66 66 66	55 55 56 56 57 57 57
5	The i 5.1 5.2 5.3	mport sy import Package 5.2.1 5.2.2 Searchin 5.3.1 5.3.2 5.3.3 5.3.4 Loading 5.4.1	ystem tlib es Regular packages Namespace packages ng The module cache Finders and loaders Import hooks The meta path g Loaders	66 66 66 66 66 66 66	55 55 56 57 57 57 57 58
5	The i 5.1 5.2 5.3	Exception Except	ystem tlib es Regular packages Namespace packages ng . The module cache Finders and loaders Import hooks The meta path g Loaders Submodules	66 66 66 66 66 66 67	55 55 56 56 57 57 57 57 58 59
5	The i 5.1 5.2 5.3	Exception Except	ystem tlib es Regular packages Namespace packages ng . The module cache Finders and loaders Import hooks The meta path g Loaders Submodules Module specs	66 66 66 66 66 66 77 77	55 55 56 66 67 67 67 67 67 68 69 70
5	The i 5.1 5.2 5.3	Exception import sy import sy import 5.2.1 5.2.2 Searchin 5.3.1 5.3.2 5.3.3 5.3.4 Loading 5.4.1 5.4.2 5.4.3 5.4.4 5.4.5 5.4.6	ystem tlib es Regular packages Namespace packages ng The module cache Finders and loaders Import hooks The meta path g Loaders Submodules Module specspath attributes on modules Module reprs Cached bytecode invalidation	66 66 66 66 66 66 77 77	55 55 55 56 56 57 57 57 58 59 70 71 71
5	The i 5.1 5.2 5.3	Exception Exception Import systems of the package 5.2.1 5.2.2 Searchin 5.3.1 5.3.2 5.3.3 5.3.4 Loading 5.4.1 5.4.2 5.4.3 5.4.4 5.4.5 5.4.6 The Patrice Import systems of the patrice in th	ystem tlib es Regular packages Namespace packages ng The module cache Finders and loaders Import hooks The meta path g Loaders Submodules Module specspath attributes on modules Module reprs Cached bytecode invalidation th Based Finder	66 66 66 66 66 67 77 77	55 55 55 56 57 57 57 57 57 57 57 57 57 57 57 57 57
5	The i 5.1 5.2 5.3	Exception import sy import sy import 5.2.1 5.2.2 Searchin 5.3.1 5.3.2 5.3.3 5.3.4 Loading 5.4.1 5.4.2 5.4.3 5.4.4 5.4.5 5.4.6	ystem tlib es Regular packages Namespace packages ng The module cache Finders and loaders Import hooks The meta path g Loaders Submodules Module specspath attributes on modules Module reprs Cached bytecode invalidation	66 66 66 66 66 67 77 77	55 55 55 56 56 57 57 57 58 59 70 71 71
5	The i 5.1 5.2 5.3	Exception Except	ystem Liib es Regular packages Namespace packages ng The module cache Finders and loaders Import hooks The meta path g Loaders Submodules Module specspath attributes on modules Module reprs Cached bytecode invalidation th Based Finder Path entry finders Path entry finder protocol	66 66 66 66 66 77 77 77 77	52 55 55 56 56 56 57 57 57 57 57 57 57 57 57 57 57 57 57
5	The i 5.1 5.2 5.3	Exception Except	ystem Liib es Regular packages Namespace packages ng The module cache Finders and loaders Import hooks The meta path g Loaders Submodules Module specspath attributes on modules Module reprs Cached bytecode invalidation th Based Finder Path entry finders	66 66 66 66 66 77 77 77 77	52 55 55 56 56 57 57 57 57 57 57 57 57 57 57 57 57 57
5	The i 5.1 5.2 5.3 5.4	Exception Import system Package 5.2.1 5.2.2 Searchin 5.3.1 5.3.2 5.3.3 5.3.4 Loading 5.4.1 5.4.2 5.4.3 5.4.4 5.4.5 5.4.6 The Path 5.5.1 5.5.2 Replacin Package	ystem Llib .es .Regular packages .Namespace packages ng .The module cache .Finders and loaders .Import hooks .The meta path .g .Loaders .Submodules .Module specspath attributes on modules .Module reprs .Cached bytecode invalidation .th Based Finder .Path entry finders .Path entry finders .Path entry finder protocol .ing the standard import system .e Relative Imports	66 66 66 66 66 67 77 77 77 77 77 77	52 55 55 56 56 56 57 57 57 57 57 57 57 57 57 57 57 57 57
5	The i 5.1 5.2 5.3 5.4 5.5	Exception Import system Package 5.2.1 5.2.2 Searchin 5.3.1 5.3.2 5.3.3 5.3.4 Loading 5.4.1 5.4.2 5.4.3 5.4.4 5.4.5 5.4.6 The Path 5.5.1 5.5.2 Replacin Package	ystem Llib es Regular packages Namespace packages ng The module cache Finders and loaders Import hooks The meta path g Loaders Submodules Module specspath attributes on modules Module reprs Cached bytecode invalidation th Based Finder Path entry finders Path entry finders Path entry finder protocol ng the standard import system	66 66 66 66 66 67 77 77 77 77 77 77	52 55 55 56 56 56 57 57 57 57 57 57 57 57 57 57 57 57 57
5	The i 5.1 5.2 5.3 5.4 5.5	Exception Import system Package 5.2.1 5.2.2 Searchin 5.3.1 5.3.2 5.3.3 5.3.4 Loading 5.4.1 5.4.2 5.4.3 5.4.4 5.4.5 5.4.6 The Path 5.5.1 5.5.2 Replacin Package	ystem Llib .es .Regular packages .Namespace packages ng .The module cache .Finders and loaders .Import hooks .The meta path .g .Loaders .Submodules .Module specspath attributes on modules .Module reprs .Cached bytecode invalidation .th Based Finder .Path entry finders .Path entry finders .Path entry finder protocol .ing the standard import system .e Relative Imports	66 66 66 66 66 77 77 77 77 77 77 77	52 55 55 56 56 56 57 75 77 75 77 75 70 70 70 71 71 71 71 71 71 71 71 71 71 71 71 71

6	Expr	ressions 77		
	6.1	Arithmetic conversions		
	6.2	Atoms		
		6.2.1 Identifiers (Names)		
		6.2.2 Literals		
		6.2.3 Parenthesized forms		
		6.2.4 Displays for lists, sets and dictionaries		
		6.2.5 List displays		
		6.2.6 Set displays		
		6.2.7 Dictionary displays		
		6.2.8 Generator expressions		
		6.2.9 Yield expressions		
	6.3	Primaries		
		6.3.1 Attribute references		
		6.3.2 Subscriptions		
		6.3.3 Slicings		
		6.3.4 Calls		
	6.4	Await expression		
	6.5	The power operator		
	6.6	1 1		
		1		
	6.7	Binary arithmetic operations		
	6.8	Shifting operations		
	6.9	Binary bitwise operations		
	6.10	Comparisons		
		6.10.1 Value comparisons		
		6.10.2 Membership test operations		
		6.10.3 Identity comparisons		
	6.11	Boolean operations		
	6.12	Assignment expressions		
	6.13	Conditional expressions		
	6.14	Lambdas		
	6.15	Expression lists		
	6.16	Evaluation order		
	6.17	Operator precedence		
7	Simp	ole statements 97		
	7.1	Expression statements		
	7.2	Assignment statements		
		7.2.1 Augmented assignment statements		
		7.2.2 Annotated assignment statements		
	7.3	The assert statement		
	7.4	The pass statement		
	7.5	The del statement		
	7.6	The return statement		
	7.7	The yield statement		
	7.8	The raise statement		
	7.9	The break statement		
	7.10	The continue statement		
	7.11	The import statement		
	7.11			
	7 12	7.11.1 Future statements		
	7.12	The global statement		
	7.13	The nonlocal statement		
	7.14	The type statement		
8	Com	pound statements 109		
U	8.1	The if statement		
	8.2			
		The while statement		
	8.3	The for statement		

	8.4	The try	statement	. 111
		8.4.1	except clause	111
		8.4.2	except * clause	112
		8.4.3	else clause	
		8.4.4	finally clause	
	8.5		th statement	
	8.6		ech statement	
	0.0	8.6.1		
			Overview	
		8.6.2	Guards	
		8.6.3	Irrefutable Case Blocks	
		8.6.4	Patterns	
	8.7	Function	n definitions	123
	8.8	Class de	finitions	125
	8.9	Coroutii	nes	127
		8.9.1	Coroutine function definition	127
		8.9.2	The async for statement	
		8.9.3	The async with statement	
	8.10		rameter lists	
	0.10	8.10.1	Generic functions	
		8.10.2	Generic classes	
		8.10.3	Generic type aliases	131
0				
9	_	level com		133
	9.1		te Python programs	
	9.2	File inp	ut	133
	9.3	Interacti	ive input	133
	9.4	Express	ion input	134
		1	•	
10	Full (Gramma	r specification	135
A	Gloss	sary		153
		t these d	ocuments	171
		t these d	ocuments utors to the Python Documentation	171
В	Abou B.1	t these d	utors to the Python Documentation	171 171
В	Abou B.1	t these d Contribu	utors to the Python Documentation	171 171 173
В	Abou B.1	t these d Contribu	utors to the Python Documentation	171 171 173
В	Abou B.1 Histo	t these d Contribu	utors to the Python Documentation	171 171 173 173
В	Abou B.1 Histo C.1	contribution of the contri	utors to the Python Documentation	171 171 173 173
В	Abou B.1 Histo C.1	Contribution or and I History Terms a C.2.1	utors to the Python Documentation	171 171 173 173 174
В	Abou B.1 Histo C.1	Contribution or and L. History Terms a C.2.1 C.2.2	utors to the Python Documentation	171 171 173 173 174 174 175
В	Abou B.1 Histo C.1	contribution of the contri	utors to the Python Documentation	171 173 173 174 174 175
В	Abou B.1 Histo C.1	contribution of the second of	utors to the Python Documentation	171 173 173 174 174 175 175
В	Abou B.1 Histo C.1	contribution of the contri	License of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMEN-	171 173 173 174 174 175 175
В	Abou B.1 Histo C.1 C.2	Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5	License of the software of the software of conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION	171 173 173 174 174 175 175
В	Abou B.1 Histo C.1	ory and I History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5	License of the software of the software of conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software	171 173 173 174 175 175 176
В	Abou B.1 Histo C.1 C.2	ry and I History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1	License of the software of conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister	171 173 173 174 175 175 176
В	Abou B.1 Histo C.1 C.2	Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2	License of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets	171 173 173 174 174 175 175 176 177 177
В	Abou B.1 Histo C.1 C.2	ry and I History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1	License of the software of conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister	171 173 173 174 174 175 175 176 177 177
В	Abou B.1 Histo C.1 C.2	Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2	License of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets	171 173 173 174 174 175 175 176 177 177 178 179
В	Abou B.1 Histo C.1 C.2	t these d Contributions and L History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2 C.3.3	cicense of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets Asynchronous socket services Cookie management	171 173 173 174 174 175 175 176 177 177 178 179
В	Abou B.1 Histo C.1 C.2	t these d Contributory and L History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2 C.3.3 C.3.4 C.3.5	cicense of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets Asynchronous socket services Cookie management Execution tracing	171 173 173 174 174 175 175 176 177 177 178 179 179
В	Abou B.1 Histo C.1 C.2	t these d Contributory and L History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2 C.3.3 C.3.4 C.3.5 C.3.6	License of the software of the software of conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets Asynchronous socket services Cookie management Execution tracing UUencode and UUdecode functions	171 173 173 174 174 175 176 177 177 178 179 179 180
В	Abou B.1 Histo C.1 C.2	t these d Contributory and I History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2 C.3.3 C.3.4 C.3.5 C.3.6 C.3.7	License of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets Asynchronous socket services Cookie management Execution tracing UUencode and UUdecode functions XML Remote Procedure Calls	171 173 173 174 174 175 176 177 177 178 179 179 180 181
В	Abou B.1 Histo C.1 C.2	t these d Contribution ory and I History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2 C.3.3 C.3.4 C.3.5 C.3.6 C.3.7 C.3.8	License of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets Asynchronous socket services Cookie management Execution tracing UUencode and UUdecode functions XML Remote Procedure Calls test_epoll	171 173 173 174 174 175 175 177 177 177 178 179 179 180 181 181
В	Abou B.1 Histo C.1 C.2	t these d Contribution ory and L History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licensee C.3.1 C.3.2 C.3.3 C.3.4 C.3.5 C.3.6 C.3.7 C.3.8 C.3.9	License of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets Asynchronous socket services Cookie management Execution tracing UUencode and UUdecode functions XML Remote Procedure Calls test_epoll Select kqueue	171 173 173 174 174 175 175 176 177 177 178 179 180 181 181 182
В	Abou B.1 Histo C.1 C.2	t these d Contribution ory and L History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2 C.3.3 C.3.4 C.3.5 C.3.6 C.3.7 C.3.8 C.3.9 C.3.10	License of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets Asynchronous socket services Cookie management Execution tracing UUencode and UUdecode functions XML Remote Procedure Calls test_epoll Select kqueue SipHash24	171 173 174 174 175 175 176 177 177 177 178 179 180 181 181 182 182
В	Abou B.1 Histo C.1 C.2	t these d Contribution ory and I History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2 C.3.3 C.3.4 C.3.5 C.3.6 C.3.7 C.3.8 C.3.9 C.3.10 C.3.11	cicense of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets Asynchronous socket services Cookie management Execution tracing UUencode and UUdecode functions XML Remote Procedure Calls test_epoll Select kqueue SipHash24 strtod and dtoa	171 173 173 174 174 175 176 177 177 178 179 180 181 181 182 182 183
В	Abou B.1 Histo C.1 C.2	t these d Contribution ory and L History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2 C.3.3 C.3.4 C.3.5 C.3.6 C.3.7 C.3.8 C.3.9 C.3.10 C.3.11 C.3.12	License of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets Asynchronous socket services Cookie management Execution tracing UUencode and UUdecode functions XML Remote Procedure Calls test_epoll Select kqueue SipHash24 strtod and dtoa OpenSSL	171 173 173 174 175 176 177 177 177 179 180 181 182 182 183 183
В	Abou B.1 Histo C.1 C.2	t these d Contribution ory and I History Terms a C.2.1 C.2.2 C.2.3 C.2.4 C.2.5 Licenses C.3.1 C.3.2 C.3.3 C.3.4 C.3.5 C.3.6 C.3.7 C.3.8 C.3.9 C.3.10 C.3.11	cicense of the software nd conditions for accessing or otherwise using Python PSF LICENSE AGREEMENT FOR PYTHON 3.13.0 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION s and Acknowledgements for Incorporated Software Mersenne Twister Sockets Asynchronous socket services Cookie management Execution tracing UUencode and UUdecode functions XML Remote Procedure Calls test_epoll Select kqueue SipHash24 strtod and dtoa	171 173 173 174 175 176 177 177 177 179 180 181 182 182 183 183

In	dex		195
D	Copyright		193
	C.3.21	Global Unbounded Sequences (GUS)	190
	C.3.20	asyncio	190
		mimalloc	
	C.3.18	W3C C14N test suite	189
	C.3.17	libmpdec	188
	C.3.16	cfuhash	188
		zlib	

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

CONTENTS 1

2 CONTENTS

INTRODUCTION

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

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

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

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

1.1 Alternate Implementations

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

Known implementations include:

CPython

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

Jython

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

Python for .NET

This implementation actually uses the CPython implementation, but is a managed .NET application and makes .NET libraries available. It was created by Brian Lloyd. For more information, see the Python for .NET home page.

IronPvthon

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

PyPy

An implementation of Python written completely in Python. It supports several advanced features not found in other implementations like stackless support and a Just in Time compiler. One of the goals of the project is

to encourage experimentation with the language itself by making it easier to modify the interpreter (since it is written in Python). Additional information is available on the PyPy project's home page.

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

1.2 Notation

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

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

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

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

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

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

CHAPTER

TWO

LEXICAL ANALYSIS

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

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

2.1 Line structure

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

2.1.1 Logical lines

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

2.1.2 Physical lines

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

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

2.1.3 Comments

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

2.1.4 Encoding declarations

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

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

which is recognized also by GNU Emacs, and

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

which is recognized by Bram Moolenaar's VIM.

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

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

2.1.5 Explicit line joining

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

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

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

2.1.6 Implicit line joining

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

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

2.1.7 Blank lines

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

2.1.8 Indentation

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

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

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

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

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

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

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

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

The following example shows various indentation errors:

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

2.1.9 Whitespace between tokens

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

2.2 Other tokens

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

2.2. Other tokens 7

2.3 Identifiers and keywords

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

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

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

Identifiers are unlimited in length. Case is significant.

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

The Unicode category codes mentioned above stand for:

- *Lu* uppercase letters
- Ll lowercase letters
- Lt titlecase letters
- Lm modifier letters
- Lo other letters
- Nl letter numbers
- *Mn* nonspacing marks
- Mc spacing combining marks
- Nd decimal numbers
- Pc connector punctuations
- Other_ID_Start explicit list of characters in PropList.txt to support backwards compatibility
- Other_ID_Continue likewise

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

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

2.3.1 Keywords

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

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

2.3.2 Soft Keywords

Added in version 3.10.

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

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

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

Changed in version 3.12: type is now a soft keyword.

2.3.3 Reserved classes of identifiers

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

- Not imported by from module import *.
- In a case pattern within a match statement, _ is a soft keyword that denotes a wildcard.

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

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

1 Note

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

It is also commonly used for unused variables.

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

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

2.4 Literals

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

2.4.1 String and Bytes literals

String literals are described by the following lexical definitions:

2.4. Literals 9

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

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

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

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

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

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

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

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

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

Escape sequences

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

Escape Sequence	Meaning	Notes
\ <newline></newline>	Backslash and newline ignored	(1)
\\	Backslash (\)	
\'	Single quote (')	
\"	Double quote (")	
\a	ASCII Bell (BEL)	
\b	ASCII Backspace (BS)	
\f	ASCII Formfeed (FF)	
\n	ASCII Linefeed (LF)	
\r	ASCII Carriage Return (CR)	
\t	ASCII Horizontal Tab (TAB)	
\v	ASCII Vertical Tab (VT)	
\000	Character with octal value ooo	(2,4)
\xhh	Character with hex value hh	(3,4)

Escape sequences only recognized in string literals are:

Escape Sequence	Meaning	Notes
\N{name}	Character named <i>name</i> in the Unicode database	(5)
\uxxxx	Character with 16-bit hex value xxxx	(6)
\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx	(7)

Notes:

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

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

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

- (2) As in Standard C, up to three octal digits are accepted.
 - Changed in version 3.11: Octal escapes with value larger than 00377 produce a DeprecationWarning.
 - Changed in version 3.12: Octal escapes with value larger than 0o377 produce a SyntaxWarning. In a future Python version they will be eventually a SyntaxError.
- (3) Unlike in Standard C, exactly two hex digits are required.
- (4) In a bytes literal, hexadecimal and octal escapes denote the byte with the given value. In a string literal, these escapes denote a Unicode character with the given value.
- (5) Changed in version 3.3: Support for name aliases¹ has been added.
- (6) Exactly four hex digits are required.
- (7) Any Unicode character can be encoded this way. Exactly eight hex digits are required.

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

Changed in version 3.6: Unrecognized escape sequences produce a DeprecationWarning.

Changed in version 3.12: Unrecognized escape sequences produce a SyntaxWarning. In a future Python version they will be eventually a SyntaxError.

2.4. Literals

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

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

2.4.2 String literal concatenation

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

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

2.4.3 f-strings

Added in version 3.6.

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

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

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

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

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

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

Changed in version 3.12: Prior to Python 3.12, comments were not allowed inside f-string replacement fields.

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

Added in version 3.8: The equal sign '='.

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

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

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

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

Some examples of formatted string literals:

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

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

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

2.4. Literals 13

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

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

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

Changed in version 3.12: Prior to Python 3.12, backslashes were not permitted inside an f-string replacement field.

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

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

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

2.4.4 Numeric literals

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

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

2.4.5 Integer literals

Integer literals are described by the following lexical definitions:

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

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

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

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

Some examples of integer literals:

```
7 2147483647 00177 0b100110111
3 79228162514264337593543950336 00377 0xdeadbeef
100_000_000_000 0b_1110_0101
```

Changed in version 3.6: Underscores are now allowed for grouping purposes in literals.

2.4.6 Floating-point literals

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

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

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

Some examples of floating-point literals:

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

Changed in version 3.6: Underscores are now allowed for grouping purposes in literals.

2.4.7 Imaginary literals

Imaginary literals are described by the following lexical definitions:

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

An imaginary literal yields a complex number with a real part of 0.0. Complex numbers are represented as a pair of floating-point numbers and have the same restrictions on their range. To create a complex number with a nonzero real part, add a floating-point number to it, e.g., $(3+4\frac{1}{2})$. Some examples of imaginary literals:

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

2.5 Operators

The following tokens are operators:

```
    +
    -
    *
    *
    //
    %
    @

    <</td>
    >>
    &
    .
    :=

    <</td>
    >
    =
    !=
```

2.6 Delimiters

The following tokens serve as delimiters in the grammar:

```
( ) [ ] { } 

, : ! . ; @ = 

-> += -= *= /= //= %= 

@= &= |= ^= >>= <<= **=
```

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

2.5. Operators 15

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

· " # \

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

\$? `

CHAPTER

THREE

DATA MODEL

3.1 Objects, values and types

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

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

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

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

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

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

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

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

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

Some objects contain references to other objects; these are called *containers*. Examples of containers are tuples, lists and dictionaries. The references are part of a container's value. In most cases, when we talk about the value of a container, we imply the values, not the identities of the contained objects; however, when we talk about the mutability

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

of a container, only the identities of the immediately contained objects are implied. So, if an immutable container (like a tuple) contains a reference to a mutable object, its value changes if that mutable object is changed.

Types affect almost all aspects of object behavior. Even the importance of object identity is affected in some sense: for immutable types, operations that compute new values may actually return a reference to any existing object with the same type and value, while for mutable objects this is not allowed. For example, after a = 1; b = 1, a and b may or may not refer to the same object with the value one, depending on the implementation. This is because int is an immutable type, so the reference to 1 can be reused. This behaviour depends on the implementation used, so should not be relied upon, but is something to be aware of when making use of object identity tests. However, after c = [1]; d = [1], c and d are guaranteed to refer to two different, unique, newly created empty lists. (Note that e = f = [1] assigns the *same* object to both e and f.)

3.2 The standard type hierarchy

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

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

3.2.1 None

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

3.2.2 NotImplemented

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

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

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

3.2.3 Ellipsis

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

3.2.4 numbers.Number

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

The string representations of the numeric classes, computed by <u>__repr__()</u> and <u>__str__()</u>, have the following properties:

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

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

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

numbers.Integral

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



1 Note

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

There are two types of integers:

Integers (int)

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

Booleans (bool)

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

numbers.Real (float)

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

numbers.Complex (complex)

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

3.2.5 Sequences

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

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

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

Sequences are distinguished according to their mutability:

Immutable sequences

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

The following types are immutable sequences:

Strings

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

Tuples

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

Bytes

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

Mutable sequences

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



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

There are currently two intrinsic mutable sequence types:

Lists

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

Byte Arrays

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

3.2.6 Set types

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

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

There are currently two intrinsic set types:

Sets

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

Frozen sets

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

3.2.7 Mappings

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

There is currently a single intrinsic mapping type:

Dictionaries

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

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

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

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

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

3.2.8 Callable types

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

User-defined functions

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

Special read-only attributes

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

Special writable attributes

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

Attribute	Meaning
functiondoc	The function's documentation string, or None if unavailable.
functionname	The function's name. See also:name attributes.
functionqualname	The function's <i>qualified name</i> . See also:qualname attributes. Added in version 3.3.
functionmodule	The name of the module the function was defined in, or None if unavailable.
functiondefaults	A tuple containing default <i>parameter</i> values for those parameters that have defaults, or None if no parameters have a default value.
functioncode	The <i>code object</i> representing the compiled function body.
functiondict	The namespace supporting arbitrary function attributes. See also:dict attributes.
functionannotations	A dictionary containing annotations of <i>parameters</i> . The keys of the dictionary are the parameter names, and 'return' for the return annotation, if provided. See also: annotations-howto.
functionkwdefaults	A dictionary containing defaults for keyword-only parameters.
functiontype_params	A tuple containing the <i>type parameters</i> of a <i>generic function</i> . Added in version 3.12.

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

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

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

Instance methods

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

Special read-only attributes:

methodself	Refers to the class instance object to which the method is <i>bound</i>
methodfunc	Refers to the original function object
methoddoc	The method's documentation (same as methodfuncdoc). A string if the original function had a docstring, else None.
methodname	The name of the method (same as methodfuncname)
methodmodule	The name of the module the method was defined in, or None if unavailable.

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

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

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

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

When an instance method object is called, the underlying function ($_func_$) is called, inserting the class instance ($_self_$) in front of the argument list. For instance, when c is a class which contains a definition for a function c (), and c is an instance of c, calling c f (1) is equivalent to calling c f (c, 1).

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

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

Generator functions

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

Coroutine functions

A function or method which is defined using <code>async def</code> is called a *coroutine function*. Such a function, when called, returns a *coroutine* object. It may contain <code>await</code> expressions, as well as <code>async with</code> and <code>async for</code> statements. See also the *Coroutine Objects* section.

Asynchronous generator functions

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

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

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

Built-in functions

A built-in function object is a wrapper around a C function. Examples of built-in functions are len() and math. sin() (math is a standard built-in module). The number and type of the arguments are determined by the C function. Special read-only attributes:

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

Built-in methods

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

Classes

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

Class Instances

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

3.2.9 Modules

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

Attribute assignment updates the module's namespace dictionary, e.g., $m \cdot x = 1$ is equivalent to $m \cdot \underline{\quad}$ dict $\underline{\quad}$ ["x"] = 1.

Import-related attributes on module objects

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

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

Caution

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

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

```
module.__name__
```

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

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

```
module.__spec__
```

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

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

Added in version 3.4.

```
module.__package__
```

The package a module belongs to.

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

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

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

Changed in version 3.4: This attribute now defaults to None for modules created dynamically using the types. Module Type constructor. Previously the attribute was optional.

Changed in version 3.6: The value of __package__ is expected to be the same as __spec__.parent. __package__ is now only used as a fallback during import resolution if __spec__.parent is not defined.

Changed in version 3.10: ImportWarning is raised if an import resolution falls back to __package__ instead of __spec__.parent.

Changed in version 3.12: Raise DeprecationWarning instead of ImportWarning when falling back to __package__ during import resolution.

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

```
module.__loader__
```

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

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

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

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

Changed in version 3.4: This attribute now defaults to None for modules created dynamically using the types. ModuleType constructor. Previously the attribute was optional.

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

```
module.__path__
```

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

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

module.__file__

module. cached

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

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

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

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

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

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

Other writable attributes on module objects

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

```
module.__doc__
```

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

```
module.__annotations__
```

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

Module dictionaries

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

```
module. dict
```

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

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

3.2.10 Custom classes

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

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

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

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

Special attributes

Attribute	Meaning
typename	The class's name. See also:name attributes.
typequalname	The class's <i>qualified name</i> . See also:qualname attributes.
typemodule	The name of the module in which the class was defined.
typedict	A mapping proxy providing a read-only view of the class's namespace. See also:dict attributes.
typebases	A tuple containing the class's bases. In most cases, for a class defined as class X(A, B, C), X. bases will be exactly equal to (A, B, C).
typedoc	The class's documentation string, or None if undefined. Not inherited by subclasses.
typeannotations	A dictionary containing <i>variable annotations</i> collected during class body execution. For best practices on working withannotations, please see annotations-howto.
	* Caution
	Accessing theannotations attribute of a class object directly may yield incorrect results in the presence of metaclasses. In addition, the attribute may not exist for some classes. Use inspect.get_annotations() to retrieve class annotations safely.
typetype_params	A tuple containing the <i>type parameters</i> of a <i>generic class</i> . Added in version 3.12.
typestatic_attributes	A tuple containing names of attributes of this class which are assigned through self.X from any function in its body. Added in version 3.13.
typefirstlineno	The line number of the first line of the class definition, including decorators. Setting themodule attribute removes thefirstlineno item from the type's dictionary. Added in version 3.13.
typemro	The tuple of classes that are considered when looking for base classes during method resolution.

Special methods

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

 $\verb|type.mro|()|$

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

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

```
type.__subclasses__()
```

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

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

3.2.11 Class instances

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

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

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

Special attributes

```
The class to which a class instance belongs.

object.__dict__
```

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

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

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

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

3.2.13 Internal types

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

Code objects

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

Special read-only attributes

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

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

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

Other bits in co_flags are reserved for internal use.

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

Methods on code objects

```
codeobject.co_positions()
```

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

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

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

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

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

Added in version 3.11.

1 Note

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

codeobject.co_lines()

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

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

The items yielded will have the following properties:

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

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

Added in version 3.10.



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

The PEP that introduced the co_lines() method.

codeobject.replace(**kwargs)

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

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

Added in version 3.8.

Frame objects

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

Special read-only attributes

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

Special writable attributes

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

Frame object methods

Frame objects support one method:

frame.clear()

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

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

Added in version 3.4.

Changed in version 3.13: Attempting to clear a suspended frame raises RuntimeError (as has always been the case for executing frames).

Traceback objects

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

Changed in version 3.7: Traceback objects can now be explicitly instantiated from Python code.

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

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

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

Special read-only attributes:

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

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

traceback.tb_next

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

Changed in version 3.7: This attribute is now writable

Slice objects

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

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

Slice objects support one method:

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

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

Static method objects

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

Class method objects

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

3.3 Special method names

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

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

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

3.3.1 Basic customization

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

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

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

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

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

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

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

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

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

```
object.__del__(self)
```

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

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

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

1 Note

 $\texttt{del} \times \texttt{doesn't}$ directly call \times . __del__ () — the former decrements the reference count for \times by one, and the latter is only called when \times 's reference count reaches zero.

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

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

→ See also

Documentation for the gc module.

Warning

Due to the precarious circumstances under which $__{del}$ () methods are invoked, exceptions that occur during their execution are ignored, and a warning is printed to sys.stderr instead. In particular:

- ___del___() can be invoked when arbitrary code is being executed, including from any arbitrary thread. If ___del___() needs to take a lock or invoke any other blocking resource, it may deadlock as the resource may already be taken by the code that gets interrupted to execute ___del___().
- ___del___() can be executed during interpreter shutdown. As a consequence, the global variables it needs to access (including other modules) may already have been deleted or set to None. Python guarantees that globals whose name begins with a single underscore are deleted from their module before other globals are deleted; if no other references to such globals exist, this may help in assuring that imported modules are still available at the time when the ___del___() method is called.

```
object.__repr__(self)
```

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

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

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

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

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

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

```
object.__bytes__(self)
```

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

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

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

See formatspec for a description of the standard formatting syntax.

The return value must be a string object.

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

Changed in version 3.4: The __format__ method of object itself raises a TypeError if passed any non-empty string.

Changed in version 3.7: object.__format__(x, '') is now equivalent to str(x) rather than format(str(x), '').

```
object.__lt__ (self, other)
object.__le__ (self, other)
object.__eq__ (self, other)
object.__gt__ (self, other)
object.__gt__ (self, other)
```

These are the so-called "rich comparison" methods. The correspondence between operator symbols and method names is as follows: x < y calls $x . __lt __(y)$, x <= y calls $x . __lt __(y)$, x == y calls $x . __lt __(y)$, x == y calls $x . __lt __(y)$, x > y calls $x . __lt __(y)$, and x >= y calls $x . __lt __(y)$.

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

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

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

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

There are no swapped-argument versions of these methods (to be used when the left argument does not support the operation but the right argument does); rather, __1t__ () and __gt__ () are each other's reflection, __1e__ () and __ge__ () are each other's reflection, and __eq__ () and __ne__ () are their own reflection. If the operands are of different types, and the right operand's type is a direct or indirect subclass of the left operand's type, the reflected method of the right operand has priority, otherwise the left operand's method has priority. Virtual subclassing is not considered.

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

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

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

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

1 Note

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

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

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

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

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

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

1 Note

By default, the __hash__() values of str and bytes objects are "salted" with an unpredictable random value. Although they remain constant within an individual Python process, they are not predictable between repeated invocations of Python.

This is intended to provide protection against a denial-of-service caused by carefully chosen inputs that exploit the worst case performance of a dict insertion, $O(n^2)$ complexity. See http://ocert.org/advisories/ocert-2011-003.html for details.

Changing hash values affects the iteration order of sets. Python has never made guarantees about this ordering (and it typically varies between 32-bit and 64-bit builds).

See also PYTHONHASHSEED.

Changed in version 3.3: Hash randomization is enabled by default.

```
object.__bool__(self)
```

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

3.3.2 Customizing attribute access

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

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

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

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

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

object.__getattribute__(self, name)

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

1 Note

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

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

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

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

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

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

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

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

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

```
object.__dir__(self)
```

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

Customizing module attribute access

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

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

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

```
import sys
from types import ModuleType

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

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

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

1 Note

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

Changed in version 3.5: __class__ module attribute is now writable.

Added in version 3.7: __getattr__ and __dir__ module attributes.

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

Implementing Descriptors

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

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

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

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

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

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

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

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

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

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

Instances of descriptors may also have the __objclass__ attribute present:

```
object.__objclass__
```

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

Invoking Descriptors

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

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

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

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

Direct Call

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

Instance Binding

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

Class Binding

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

Super Binding

A dotted lookup such as $super(A, a) . x searches a . __class __ . __mro__ for a base class B following A and then returns B . __dict __ ['x'] . __get __ (a, A) . If not a descriptor, x is returned unchanged.$

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

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

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

slots

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

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

```
object.__slots__
```

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

Notes on using __*slots*__:

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

3.3.3 Customizing class creation

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

```
classmethod object.__init_subclass__(cls)
```

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

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

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

(continues on next page)

(continued from previous page)

```
cls.default_name = default_name

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

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

1 Note

The metaclass hint metaclass is consumed by the rest of the type machinery, and is never passed to __init_subclass__ implementations. The actual metaclass (rather than the explicit hint) can be accessed as type (cls).

Added in version 3.6.

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

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

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

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

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

```
class A:
    pass

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

See Creating the class object for more details.

Added in version 3.6.

Metaclasses

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

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

```
class Meta(type):
    pass

class MyClass(metaclass=Meta):
    pass

class MySubclass(MyClass):
    pass
```

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

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

- MRO entries are resolved;
- the appropriate metaclass is determined;
- the class namespace is prepared;
- the class body is executed;
- the class object is created.

Resolving MRO entries

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

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

```
types.resolve_bases()
          Dynamically resolve bases that are not instances of type.

types.get_original_bases()
          Retrieve a class's "original bases" prior to modifications by __mro_entries__().

PEP 560
          Core support for typing module and generic types.
```

Determining the appropriate metaclass

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

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

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

Preparing the class namespace

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

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

```
→ See also
```

PEP 3115 - Metaclasses in Python 3000

Introduced the __prepare__ namespace hook

Executing the class body

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

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

Creating the class object

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

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

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

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

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

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

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

PEP 3135 - New super Describes the implicit __class__ closure reference

Uses for metaclasses

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

3.3.4 Customizing instance and subclass checks

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

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

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

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

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

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

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

♂ See also

PEP 3119 - Introducing Abstract Base Classes

Includes the specification for customizing <code>isinstance()</code> and <code>issubclass()</code> behavior through <code>__instancecheck__()</code> and <code>__subclasscheck__()</code>, with motivation for this functionality in the context of adding Abstract Base Classes (see the abc module) to the language.

3.3.5 Emulating generic types

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

See also

PEP 484 - Type Hints

Introducing Python's framework for type annotations

Generic Alias Types

Documentation for objects representing parameterized generic classes

Generics, user-defined generics and typing. Generic

Documentation on how to implement generic classes that can be parameterized at runtime and understood by static type-checkers.

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

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

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

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

The purpose of class getitem

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

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

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

```
__class_getitem__ versus __getitem__
```

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

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

```
from inspect import isclass
def subscribe(obj, x):
    """Return the result of the expression 'obj[x]'"""
   class_of_obj = type(obj)
    # If the class of obj defines __getitem__,
    # call class_of_obj.__getitem__(obj, x)
    if hasattr(class_of_obj, '__getitem__'):
        return class_of_obj.__getitem__(obj, x)
    # Else, if obj is a class and defines __class_getitem__,
    # call obj.__class_getitem__(x)
    elif isclass(obj) and hasattr(obj, '__class_getitem__'):
        return obj.__class_getitem__(x)
    # Else, raise an exception
    else:
       raise TypeError(
           f"'{class_of_obj.__name__}' object is not subscriptable"
        )
```

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

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

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

```
>>> from enum import Enum
>>> class Menu (Enum):
... """A breakfast menu"""

(continues on next page)
```

(continued from previous page)

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

```
PEP 560 - Core Support for typing module and generic types

Introducing __class_getitem__(), and outlining when a subscription results in __class_getitem__() being called instead of __getitem__()
```

3.3.6 Emulating callable objects

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

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

3.3.7 Emulating container types

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

```
object.__len__(self)
```

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

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

```
object.__length_hint__(self)
```

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

Added in version 3.4.

1 Note

Slicing is done exclusively with the following three methods. A call like

a[1:2] = b

is translated to

a[slice(1, 2, None)] = b

and so forth. Missing slice items are always filled in with None.

object.__getitem__(self, key)

Called to implement evaluation of <code>self[key]</code>. For *sequence* types, the accepted keys should be integers. Optionally, they may support <code>slice</code> objects as well. Negative index support is also optional. If *key* is of an inappropriate type, <code>TypeError</code> may be raised; if *key* is a value outside the set of indexes for the sequence (after any special interpretation of negative values), <code>IndexError</code> should be raised. For *mapping* types, if *key* is missing (not in the container), <code>KeyError</code> should be raised.

1 Note

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

1 Note

When *subscripting* a *class*, the special class method __class_getitem__() may be called instead of __getitem__(). See __class_getitem__ versus __getitem__ for more details.

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

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

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

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

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

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

```
object.__iter__(self)
```

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

```
object.__reversed__(self)
```

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

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

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

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

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

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

3.3.8 Emulating numeric types

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

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

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

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

```
object.__radd__ (self, other)
object.__rsub__ (self, other)
object.__rmul__ (self, other)
object.__rmatmul__ (self, other)
object.__rtruediv__ (self, other)
object.__rfloordiv__ (self, other)
```

```
object.__rmod__ (self, other)
object.__rdivmod__ (self, other)
object.__rpow__ (self, other[, modulo])
object.__rlshift__ (self, other)
object.__rshift__ (self, other)
object.__rand__ (self, other)
object.__rxor__ (self, other)
object.__ror__ (self, other)
```

These methods are called to implement the binary arithmetic operations (+, -, *, @, /, //, %, divmod(), pow(), **, <<, >>, &, ^, |) with reflected (swapped) operands. These functions are only called if the left operand does not support the corresponding operation³ and the operands are of different types.⁴ For instance, to evaluate the expression x - y, where y is an instance of a class that has an $_rsub_()$ method, type(y). $_rsub_(y, x)$ is called if type(x). $_sub_(x, y)$ returns NotImplemented.

Note that ternary pow() will not try calling $\underline{rpow}()$ (the coercion rules would become too complicated).

1 Note

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

```
object.__iadd__ (self, other)
object.__isub__ (self, other)
object.__imul__ (self, other)
object.__imatmul__ (self, other)
object.__itruediv__ (self, other)
object.__ifloordiv__ (self, other)
object.__ipow__ (self, other)
object.__ipow__ (self, other[, modulo])
object.__ishift__ (self, other)
object.__ishift__ (self, other)
object.__iand__ (self, other)
object.__ixor__ (self, other)
object.__ior__ (self, other)
```

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

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

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

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

```
object.__invert__(self)
```

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

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

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

```
object.__index__(self)
```

Called to implement operator.index(), and whenever Python needs to losslessly convert the numeric object to an integer object (such as in slicing, or in the built-in bin(), hex() and oct() functions). Presence of this method indicates that the numeric object is an integer type. Must return an integer.

If __int__(), __float__() and __complex__() are not defined then corresponding built-in functions int(), float() and complex() fall back to __index__().

```
object.__round__(self[, ndigits])
object.__trunc__(self)
object.__floor__(self)
object.__ceil__(self)
```

Called to implement the built-in function round() and math functions trunc(), floor() and ceil(). Unless *ndigits* is passed to __round__() all these methods should return the value of the object truncated to an Integral (typically an int).

The built-in function int () falls back to __trunc__() if neither __int__() nor __index__() is defined.

Changed in version 3.11: The delegation of int () to __trunc__() is deprecated.

3.3.9 With Statement Context Managers

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

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

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

```
object.__enter__(self)
```

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

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

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

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

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

♂ See also

PEP 343 - The "with" statement

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

3.3.10 Customizing positional arguments in class pattern matching

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

```
object.__match_args___
```

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

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

Added in version 3.10.

```
See also
```

PEP 634 - Structural Pattern Matching

The specification for the Python match statement.

3.3.11 Emulating buffer types

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

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

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

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

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

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

Added in version 3.12.

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

Introduces the Python __buffer__ and __release_buffer__ methods.

collections.abc.Buffer

ABC for buffer types.
```

3.3.12 Special method lookup

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

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

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

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

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

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

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

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

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

3.4 Coroutines

3.4.1 Awaitable Objects

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



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

```
object.__await__(self)
```

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



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

Added in version 3.5.

See also

PEP 492 for additional information about awaitable objects.

3.4.2 Coroutine Objects

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

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

Changed in version 3.5.2: It is a RuntimeError to await on a coroutine more than once.

```
coroutine.send(value)
```

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

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

Raises the specified exception in the coroutine. This method delegates to the <code>throw()</code> method of the iterator that caused the coroutine to suspend, if it has such a method. Otherwise, the exception is raised at the suspension point. The result (return value, <code>StopIteration</code>, or other exception) is the same as when iterating over the <code>__await__()</code> return value, described above. If the exception is not caught in the coroutine, it propagates back to the caller.

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

3.4. Coroutines 55

```
coroutine.close()
```

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

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

3.4.3 Asynchronous Iterators

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

Asynchronous iterators can be used in an async for statement.

The object class itself does not provide these methods.

```
object.__aiter__(self)
```

Must return an asynchronous iterator object.

```
object.__anext__(self)
```

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

An example of an asynchronous iterable object:

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

def __aiter__(self):
    return self

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

Added in version 3.5.

Changed in version 3.7: Prior to Python 3.7, __aiter__() could return an awaitable that would resolve to an asynchronous iterator.

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

3.4.4 Asynchronous Context Managers

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

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

The object class itself does not provide these methods.

```
object.__aenter__(self)
```

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

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

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

An example of an asynchronous context manager class:

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

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

Added in version 3.5.

3.4. Coroutines 57

EXECUTION MODEL

4.1 Structure of a program

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

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

4.2 Naming and binding

4.2.1 Binding of names

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

The following constructs bind names:

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

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

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

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

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

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

4.2.2 Resolution of names

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

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

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

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

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

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

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

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

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

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

However, the following will succeed:

```
class A:
    type Alias = Nested
    class Nested: pass
print(A.Alias.__value__) # <type 'A.Nested'>
```

4.2.3 Annotation scopes

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

Annotation scopes are used in the following contexts:

- Type parameter lists for generic type aliases.
- Type parameter lists for *generic functions*. A generic function's annotations are executed within the annotation scope, but its defaults and decorators are not.
- Type parameter lists for *generic classes*. A generic class's base classes and keyword arguments are executed within the annotation scope, but its decorators are not.
- The bounds, constraints, and default values for type parameters (lazily evaluated).
- The value of type aliases (lazily evaluated).

Annotation scopes differ from function scopes in the following ways:

- Annotation scopes have access to their enclosing class namespace. If an annotation scope is immediately
 within a class scope, or within another annotation scope that is immediately within a class scope, the code in
 the annotation scope can use names defined in the class scope as if it were executed directly within the class
 body. This contrasts with regular functions defined within classes, which cannot access names defined in the
 class scope.
- Expressions in annotation scopes cannot contain *yield*, yield from, *await*, or := expressions. (These expressions are allowed in other scopes contained within the annotation scope.)
- Names defined in annotation scopes cannot be rebound with nonlocal statements in inner scopes. This includes only type parameters, as no other syntactic elements that can appear within annotation scopes can introduce new names.
- While annotation scopes have an internal name, that name is not reflected in the *qualified name* of objects defined within the scope. Instead, the __qualname__ of such objects is as if the object were defined in the enclosing scope.

Added in version 3.12: Annotation scopes were introduced in Python 3.12 as part of PEP 695.

Changed in version 3.13: Annotation scopes are also used for type parameter defaults, as introduced by PEP 696.

4.2.4 Lazy evaluation

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

Example:

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

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

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

```
from typing import Literal

type SimpleExpr = int | Parenthesized

type Parenthesized = tuple[Literal["("], Expr, Literal[")"]]

type Expr = SimpleExpr | tuple[SimpleExpr, Literal["+", "-"], Expr]
```

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

Added in version 3.12.

4.2.5 Builtins and restricted execution

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

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

4.2.6 Interaction with dynamic features

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

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

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

4.3 Exceptions

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

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

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

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

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

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

1 Note

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

See also the description of the try statement in section The try statement and raise statement in section The raise statement.

4.3. Exceptions 63

CHAPTER

FIVE

THE IMPORT SYSTEM

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

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

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

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

When a module is first imported, Python searches for the module and if found, it creates a module object¹, initializing it. If the named module cannot be found, a ModuleNotFoundError is raised. Python implements various strategies to search for the named module when the import machinery is invoked. These strategies can be modified and extended by using various hooks described in the sections below.

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

5.1 importlib

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

5.2 Packages

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

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

¹ See types.ModuleType.

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

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

5.2.1 Regular packages

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

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

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

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

5.2.2 Namespace packages

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

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

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

See also PEP 420 for the namespace package specification.

5.3 Searching

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

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

5.3.1 The module cache

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

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

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

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

5.3.2 Finders and loaders

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

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

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

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

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

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

5.3.3 Import hooks

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

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

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

5.3.4 The meta path

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

5.3. Searching 67

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

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

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

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

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

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

Changed in version 3.10: Use of find_module() by the import system now raises ImportWarning.

Changed in version 3.12: find_module() has been removed. Use find_spec() instead.

5.4 Loading

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

```
module = None
if spec.loader is not None and hasattr(spec.loader, 'create_module'):
    # It is assumed 'exec_module' will also be defined on the loader.
   module = spec.loader.create_module(spec)
if module is None:
   module = ModuleType(spec.name)
# The import-related module attributes get set here:
_init_module_attrs(spec, module)
if spec.loader is None:
    # unsupported
   raise ImportError
if spec.origin is None and spec.submodule_search_locations is not None:
    # namespace package
    sys.modules[spec.name] = module
elif not hasattr(spec.loader, 'exec_module'):
    module = spec.loader.load_module(spec.name)
    sys.modules[spec.name] = module
        spec.loader.exec_module(module)
```

(continues on next page)

(continued from previous page)

```
except BaseException:
    try:
        del sys.modules[spec.name]
    except KeyError:
        pass
    raise
return sys.modules[spec.name]
```

Note the following details:

- If there is an existing module object with the given name in sys.modules, import will have already returned it
- The module will exist in sys.modules before the loader executes the module code. This is crucial because the module code may (directly or indirectly) import itself; adding it to sys.modules beforehand prevents unbounded recursion in the worst case and multiple loading in the best.
- If loading fails, the failing module and only the failing module gets removed from sys.modules. Any module already in the sys.modules cache, and any module that was successfully loaded as a side-effect, must remain in the cache. This contrasts with reloading where even the failing module is left in sys.modules.
- After the module is created but before execution, the import machinery sets the import-related module attributes ("_init_module_attrs" in the pseudo-code example above), as summarized in a *later section*.
- Module execution is the key moment of loading in which the module's namespace gets populated. Execution is entirely delegated to the loader, which gets to decide what gets populated and how.
- The module created during loading and passed to exec_module() may not be the one returned at the end of import².

Changed in version 3.4: The import system has taken over the boilerplate responsibilities of loaders. These were previously performed by the importlib.abc.Loader.load_module() method.

5.4.1 Loaders

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

Loaders must satisfy the following requirements:

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

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

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

Added in version 3.4: The create_module() method of loaders.

Changed in version 3.4: The <code>load_module()</code> method was replaced by <code>exec_module()</code> and the import machinery assumed all the boilerplate responsibilities of loading.

5.4. Loading 69

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

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

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

- If there is an existing module object with the given name in sys.modules, the loader must use that existing module. (Otherwise, importlib.reload() will not work correctly.) If the named module does not exist in sys.modules, the loader must create a new module object and add it to sys.modules.
- The module *must* exist in sys.modules before the loader executes the module code, to prevent unbounded recursion or multiple loading.
- If loading fails, the loader must remove any modules it has inserted into sys.modules, but it must remove **only** the failing module(s), and only if the loader itself has loaded the module(s) explicitly.

Changed in version 3.5: A DeprecationWarning is raised when exec_module() is defined but create_module() is not.

Changed in version 3.6: An ImportError is raised when exec_module() is defined but create_module() is not.

Changed in version 3.10: Use of load_module() will raise ImportWarning.

5.4.2 Submodules

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

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

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

```
from .foo import Foo
```

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

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

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

5.4.3 Module specs

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

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

The module's spec is exposed as <code>module.__spec_</code>. Setting <code>__spec__</code> appropriately applies equally to modules initialized during interpreter startup. The one exception is <code>__main__</code>, where <code>__spec__</code> is set to None in some cases.

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

Added in version 3.4.

5.4.4 path attributes on modules

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

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

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

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

5.4.5 Module reprs

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

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

Here are the exact rules used:

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

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

5.4.6 Cached bytecode invalidation

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

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

Changed in version 3.7: Added hash-based .pyc files. Previously, Python only supported timestamp-based invalidation of bytecode caches.

5.4. Loading 71

5.5 The Path Based Finder

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

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

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

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

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

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

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

5.5.1 Path entry finders

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

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

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

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

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

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

If the path entry is not present in the cache, the path based finder iterates over every callable in sys.path_hooks. Each of the *path entry hooks* in this list is called with a single argument, the path entry to be searched. This callable

may either return a *path entry finder* that can handle the path entry, or it may raise ImportError. An ImportError is used by the path based finder to signal that the hook cannot find a *path entry finder* for that *path entry*. The exception is ignored and *import path* iteration continues. The hook should expect either a string or bytes object; the encoding of bytes objects is up to the hook (e.g. it may be a file system encoding, UTF-8, or something else), and if the hook cannot decode the argument, it should raise ImportError.

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

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

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

5.5.2 Path entry finder protocol

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

find_spec() takes two arguments: the fully qualified name of the module being imported, and the (optional) target module. find_spec() returns a fully populated spec for the module. This spec will always have "loader" set (with one exception).

To indicate to the import machinery that the spec represents a namespace *portion*, the path entry finder sets submodule_search_locations to a list containing the portion.

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

Older path entry finders may implement one of these two deprecated methods instead of find_spec(). The methods are still respected for the sake of backward compatibility. However, if find_spec() is implemented on the path entry finder, the legacy methods are ignored.

find_loader() takes one argument, the fully qualified name of the module being imported. find_loader() returns a 2-tuple where the first item is the loader and the second item is a namespace *portion*.

For backwards compatibility with other implementations of the import protocol, many path entry finders also support the same, traditional find_module() method that meta path finders support. However path entry finder find_module() methods are never called with a path argument (they are expected to record the appropriate path information from the initial call to the path hook).

The find_module() method on path entry finders is deprecated, as it does not allow the path entry finder to contribute portions to namespace packages. If both find_loader() and find_module() exist on a path entry finder, the import system will always call find_loader() in preference to find_module().

Changed in version 3.10: Calls to find_module() and find_loader() by the import system will raise ImportWarning.

Changed in version 3.12: find_module() and find_loader() have been removed.

5.6 Replacing the standard import system

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

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

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

5.7 Package Relative Imports

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

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

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

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

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

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

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

5.8 Special considerations for __main__

The __main__ module is a special case relative to Python's import system. As noted *elsewhere*, the __main__ module is directly initialized at interpreter startup, much like sys and builtins. However, unlike those two, it doesn't strictly qualify as a built-in module. This is because the manner in which __main__ is initialized depends on the flags and other options with which the interpreter is invoked.

```
5.8.1 __main__._spec__
```

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

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

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

- · interactive prompt
- -c option

- · running from stdin
- running directly from a source or bytecode file

Note that __main__ . __spec__ is always None in the last case, *even if* the file could technically be imported directly as a module instead. Use the -m switch if valid module metadata is desired in __main__.

Note also that even when __main__ corresponds with an importable module and __main__.__spec__ is set accordingly, they're still considered *distinct* modules. This is due to the fact that blocks guarded by if __name__ = "__main__": checks only execute when the module is used to populate the __main__ namespace, and not during normal import.

5.9 References

The import machinery has evolved considerably since Python's early days. The original specification for packages is still available to read, although some details have changed since the writing of that document.

The original specification for sys.meta_path was PEP 302, with subsequent extension in PEP 420.

PEP 420 introduced *namespace packages* for Python 3.3. **PEP 420** also introduced the find_loader() protocol as an alternative to find_module().

PEP 366 describes the addition of the __package__ attribute for explicit relative imports in main modules.

PEP 328 introduced absolute and explicit relative imports and initially proposed __name__ for semantics **PEP 366** would eventually specify for __package__.

PEP 338 defines executing modules as scripts.

PEP 451 adds the encapsulation of per-module import state in spec objects. It also off-loads most of the boilerplate responsibilities of loaders back onto the import machinery. These changes allow the deprecation of several APIs in the import system and also addition of new methods to finders and loaders.

5.9. References 75

CHAPTER

SIX

EXPRESSIONS

This chapter explains the meaning of the elements of expressions in Python.

Syntax Notes: In this and the following chapters, extended BNF notation will be used to describe syntax, not lexical analysis. When (one alternative of) a syntax rule has the form

```
name ::= othername
```

and no semantics are given, the semantics of this form of name are the same as for othername.

6.1 Arithmetic conversions

When a description of an arithmetic operator below uses the phrase "the numeric arguments are converted to a common type", this means that the operator implementation for built-in types works as follows:

- If either argument is a complex number, the other is converted to complex;
- otherwise, if either argument is a floating-point number, the other is converted to floating point;
- otherwise, both must be integers and no conversion is necessary.

Some additional rules apply for certain operators (e.g., a string as a left argument to the '%' operator). Extensions must define their own conversion behavior.

6.2 Atoms

Atoms are the most basic elements of expressions. The simplest atoms are identifiers or literals. Forms enclosed in parentheses, brackets or braces are also categorized syntactically as atoms. The syntax for atoms is:

6.2.1 Identifiers (Names)

An identifier occurring as an atom is a name. See section *Identifiers and keywords* for lexical definition and section *Naming and binding* for documentation of naming and binding.

When the name is bound to an object, evaluation of the atom yields that object. When a name is not bound, an attempt to evaluate it raises a NameError exception.

Private name mangling

When an identifier that textually occurs in a class definition begins with two or more underscore characters and does not end in two or more underscores, it is considered a *private name* of that class.

```
★ See also
The class specifications.
```

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

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

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

The transformation rule is defined as follows:

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

6.2.2 Literals

Python supports string and bytes literals and various numeric literals:

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

Evaluation of a literal yields an object of the given type (string, bytes, integer, floating-point number, complex number) with the given value. The value may be approximated in the case of floating-point and imaginary (complex) literals. See section *Literals* for details.

All literals correspond to immutable data types, and hence the object's identity is less important than its value. Multiple evaluations of literals with the same value (either the same occurrence in the program text or a different occurrence) may obtain the same object or a different object with the same value.

6.2.3 Parenthesized forms

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

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

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

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

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

6.2.4 Displays for lists, sets and dictionaries

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

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

Common syntax elements for comprehensions are:

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

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

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

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

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

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

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

Added in version 3.6: Asynchronous comprehensions were introduced.

Changed in version 3.8: yield and yield from prohibited in the implicitly nested scope.

Changed in version 3.11: Asynchronous comprehensions are now allowed inside comprehensions in asynchronous functions. Outer comprehensions implicitly become asynchronous.

6.2.5 List displays

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

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

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

6.2. Atoms 79

6.2.6 Set displays

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

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

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

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

6.2.7 Dictionary displays

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

A dictionary display yields a new dictionary object.

If a comma-separated sequence of dict items is given, they are evaluated from left to right to define the entries of the dictionary: each key object is used as a key into the dictionary to store the corresponding value. This means that you can specify the same key multiple times in the dict item list, and the final dictionary's value for that key will be the last one given.

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

Added in version 3.5: Unpacking into dictionary displays, originally proposed by PEP 448.

A dict comprehension, in contrast to list and set comprehensions, needs two expressions separated with a colon followed by the usual "for" and "if" clauses. When the comprehension is run, the resulting key and value elements are inserted in the new dictionary in the order they are produced.

Restrictions on the types of the key values are listed earlier in section *The standard type hierarchy*. (To summarize, the key type should be *hashable*, which excludes all mutable objects.) Clashes between duplicate keys are not detected; the last value (textually rightmost in the display) stored for a given key value prevails.

Changed in version 3.8: Prior to Python 3.8, in dict comprehensions, the evaluation order of key and value was not well-defined. In CPython, the value was evaluated before the key. Starting with 3.8, the key is evaluated before the value, as proposed by **PEP 572**.

6.2.8 Generator expressions

A generator expression is a compact generator notation in parentheses:

```
generator_expression ::= "(" expression comp_for ")"
```

A generator expression yields a new generator object. Its syntax is the same as for comprehensions, except that it is enclosed in parentheses instead of brackets or curly braces.

Variables used in the generator expression are evaluated lazily when the __next__ () method is called for the generator object (in the same fashion as normal generators). However, the iterable expression in the leftmost for clause is immediately evaluated, so that an error produced by it will be emitted at the point where the generator expression is defined, rather than at the point where the first value is retrieved. Subsequent for clauses and any filter condition

in the leftmost for clause cannot be evaluated in the enclosing scope as they may depend on the values obtained from the leftmost iterable. For example: (x*y for x in range(10) for y in range(x, x+10)).

The parentheses can be omitted on calls with only one argument. See section Calls for details.

To avoid interfering with the expected operation of the generator expression itself, yield and yield from expressions are prohibited in the implicitly defined generator.

If a generator expression contains either async for clauses or await expressions it is called an asynchronous generator expression. An asynchronous generator expression returns a new asynchronous generator object, which is an asynchronous iterator (see Asynchronous Iterators).

Added in version 3.6: Asynchronous generator expressions were introduced.

Changed in version 3.7: Prior to Python 3.7, asynchronous generator expressions could only appear in async def coroutines. Starting with 3.7, any function can use asynchronous generator expressions.

Changed in version 3.8: yield and yield from prohibited in the implicitly nested scope.

6.2.9 Yield expressions

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

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

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

Changed in version 3.8: Yield expressions prohibited in the implicitly nested scopes used to implement comprehensions and generator expressions.

Generator functions are described below, while asynchronous generator functions are described separately in section *Asynchronous generator functions*.

When a generator function is called, it returns an iterator known as a generator. That generator then controls the execution of the generator function. The execution starts when one of the generator's methods is called. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of $yield_list$ to the generator's caller, or None if $yield_list$ is omitted. By suspended, we mean that all local state is retained, including the current bindings of local variables, the instruction pointer, the internal evaluation stack, and the state of any exception handling. When the execution is resumed by calling one of the generator's methods, the function can proceed exactly as if the yield expression were just another external call. The value of the yield expression after resuming depends on the method which resumed the execution. If wildent (to be a considered in the proceed (to be a considered in the process (to be a considered in the

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

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

6.2. Atoms 81

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

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

Changed in version 3.3: Added yield from <expr> to delegate control flow to a subiterator.

The parentheses may be omitted when the yield expression is the sole expression on the right hand side of an assignment statement.

→ See also

PEP 255 - Simple Generators

The proposal for adding generators and the yield statement to Python.

PEP 342 - Coroutines via Enhanced Generators

The proposal to enhance the API and syntax of generators, making them usable as simple coroutines.

PEP 380 - Syntax for Delegating to a Subgenerator

The proposal to introduce the yield_from syntax, making delegation to subgenerators easy.

PEP 525 - Asynchronous Generators

The proposal that expanded on PEP 492 by adding generator capabilities to coroutine functions.

Generator-iterator methods

This subsection describes the methods of a generator iterator. They can be used to control the execution of a generator function.

Note that calling any of the generator methods below when the generator is already executing raises a ValueError exception.

```
generator.__next__()
```

Starts the execution of a generator function or resumes it at the last executed yield expression. When a generator function is resumed with a __next__() method, the current yield expression always evaluates to None. The execution then continues to the next yield expression, where the generator is suspended again, and the value of the yield_list is returned to __next__()'s caller. If the generator exits without yielding another value, a StopIteration exception is raised.

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

```
generator.send(value)
```

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

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

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

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

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

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

```
generator.close()
```

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

Changed in version 3.13: If a generator returns a value upon being closed, the value is returned by close().

Examples

Here is a simple example that demonstrates the behavior of generators and generator functions:

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

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

Asynchronous generator functions

The presence of a yield expression in a function or method defined using async def further defines the function as an asynchronous generator function.

When an asynchronous generator function is called, it returns an asynchronous iterator known as an asynchronous generator object. That object then controls the execution of the generator function. An asynchronous generator object is typically used in an async for statement in a coroutine function analogously to how a generator object would be used in a for statement.

Calling one of the asynchronous generator's methods returns an *awaitable* object, and the execution starts when this object is awaited on. At that time, the execution proceeds to the first yield expression, where it is suspended again, returning the value of <code>yield_list</code> to the awaiting coroutine. As with a generator, suspension means that all local

6.2. Atoms 83

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

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

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

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

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

Asynchronous generator-iterator methods

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

```
coroutine agen.__anext__()
```

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

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

```
coroutine agen.asend(value)
```

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

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

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

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

```
coroutine agen.aclose()
```

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

6.3 Primaries

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

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

6.3.1 Attribute references

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

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

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

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

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

6.3.2 Subscriptions

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

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

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

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

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

Changed in version 3.11: Expressions in an expression list may be starred. See PEP 646.

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

6.3. Primaries 85

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

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

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

6.3.3 Slicings

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

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

The semantics for a slicing are as follows. The primary is indexed (using the same __getitem__() method as normal subscription) with a key that is constructed from the slice list, as follows. If the slice list contains at least one comma, the key is a tuple containing the conversion of the slice items; otherwise, the conversion of the lone slice item is the key. The conversion of a slice item that is an expression is that expression. The conversion of a proper slice is a slice object (see section *The standard type hierarchy*) whose start, stop and step attributes are the values of the expressions given as lower bound, upper bound and stride, respectively, substituting None for missing expressions.

6.3.4 Calls

A call calls a callable object (e.g., a *function*) with a possibly empty series of *arguments*:

```
∷= primary "(" [argument_list [","] | comprehension] ")"
call
                    ::= positional_arguments ["," starred_and_keywords]
argument_list
                         ["," keywords_arguments]
                         | starred_and_keywords ["," keywords_arguments]
                         | keywords_arguments
positional_arguments
                         positional_item ("," positional_item) *
                    ::=
positional_item
                    ::= assignment_expression | "*" expression
"" expression | keyword_item"
                        ("," "*" expression | "," keyword_item)*
keywords_arguments
                    ::= (keyword_item | "**" expression)
                         ("," keyword_item | "," "**" expression) *
                         identifier "=" expression
keyword_item
                    ::=
```

An optional trailing comma may be present after the positional and keyword arguments but does not affect the semantics.

The primary must evaluate to a callable object (user-defined functions, built-in functions, methods of built-in objects, class objects, methods of class instances, and all objects having a __call__() method are callable). All argument expressions are evaluated before the call is attempted. Please refer to section *Function definitions* for the syntax of formal *parameter* lists.

If keyword arguments are present, they are first converted to positional arguments, as follows. First, a list of unfilled slots is created for the formal parameters. If there are N positional arguments, they are placed in the first N slots. Next, for each keyword argument, the identifier is used to determine the corresponding slot (if the identifier is the same as the first formal parameter name, the first slot is used, and so on). If the slot is already filled, a TypeError exception is raised. Otherwise, the argument is placed in the slot, filling it (even if the expression is None, it fills the slot). When all arguments have been processed, the slots that are still unfilled are filled with the corresponding default value from the function definition. (Default values are calculated, once, when the function is defined; thus, a mutable object such as a list or dictionary used as default value will be shared by all calls that don't specify an argument value for the corresponding slot; this should usually be avoided.) If there are any unfilled slots for which no default value is specified, a TypeError exception is raised. Otherwise, the list of filled slots is used as the argument list for the call.

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

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

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

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

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

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

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

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

When **expression is used, each key in this mapping must be a string. Each value from the mapping is assigned to the first formal parameter eligible for keyword assignment whose name is equal to the key. A key need not be a Python identifier (e.g. "max-temp °F" is acceptable, although it will not match any formal parameter that could be

6.3. Primaries 87

declared). If there is no match to a formal parameter the key-value pair is collected by the ** parameter, if there is one, or if there is not, a TypeError exception is raised.

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

Changed in version 3.5: Function calls accept any number of * and ** unpackings, positional arguments may follow iterable unpackings (*), and keyword arguments may follow dictionary unpackings (**). Originally proposed by **PEP 448**.

A call always returns some value, possibly None, unless it raises an exception. How this value is computed depends on the type of the callable object.

If it is—

a user-defined function:

The code block for the function is executed, passing it the argument list. The first thing the code block will do is bind the formal parameters to the arguments; this is described in section *Function definitions*. When the code block executes a *return* statement, this specifies the return value of the function call. If execution reaches the end of the code block without executing a *return* statement, the return value is None.

a built-in function or method:

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

a class object:

A new instance of that class is returned.

a class instance method:

The corresponding user-defined function is called, with an argument list that is one longer than the argument list of the call: the instance becomes the first argument.

a class instance:

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

6.4 Await expression

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

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

Added in version 3.5.

6.5 The power operator

The power operator binds more tightly than unary operators on its left; it binds less tightly than unary operators on its right. The syntax is:

```
power ::= (await_expr | primary) ["**" u_expr]
```

Thus, in an unparenthesized sequence of power and unary operators, the operators are evaluated from right to left (this does not constrain the evaluation order for the operands): -1**2 results in -1.

The power operator has the same semantics as the built-in pow() function, when called with two arguments: it yields its left argument raised to the power of its right argument. The numeric arguments are first converted to a common type, and the result is of that type.

For int operands, the result has the same type as the operands unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, 10**2 returns 100, but 10**-2 returns 0.01.

Raising 0.0 to a negative power results in a ZeroDivisionError. Raising a negative number to a fractional power results in a complex number. (In earlier versions it raised a ValueError.)

This operation can be customized using the special __pow__ () and __rpow__ () methods.

6.6 Unary arithmetic and bitwise operations

All unary arithmetic and bitwise operations have the same priority:

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

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

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

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

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

6.7 Binary arithmetic operations

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

The * (multiplication) operator yields the product of its arguments. The arguments must either both be numbers, or one argument must be an integer and the other must be a sequence. In the former case, the numbers are converted to a common type and then multiplied together. In the latter case, sequence repetition is performed; a negative repetition factor yields an empty sequence.

This operation can be customized using the special __mul__() and __rmul__() methods.

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

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

Added in version 3.5.

The / (division) and // (floor division) operators yield the quotient of their arguments. The numeric arguments are first converted to a common type. Division of integers yields a float, while floor division of integers results in an integer; the result is that of mathematical division with the 'floor' function applied to the result. Division by zero raises the <code>ZeroDivisionError</code> exception.

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

The % (modulo) operator yields the remainder from the division of the first argument by the second. The numeric arguments are first converted to a common type. A zero right argument raises the <code>ZeroDivisionError</code> exception. The arguments may be floating-point numbers, e.g., 3.14%0.7 equals 0.34 (since 3.14 equals 4*0.7 + 0.34.)

The modulo operator always yields a result with the same sign as its second operand (or zero); the absolute value of the result is strictly smaller than the absolute value of the second operand ¹.

The floor division and modulo operators are connected by the following identity: x = (x//y) * y + (x*y). Floor division and modulo are also connected with the built-in function divmod (): divmod (x, y) = (x//y, x*y).

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

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

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

The + (addition) operator yields the sum of its arguments. The arguments must either both be numbers or both be sequences of the same type. In the former case, the numbers are converted to a common type and then added together. In the latter case, the sequences are concatenated.

This operation can be customized using the special __add__() and __radd__() methods.

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

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

6.8 Shifting operations

The shifting operations have lower priority than the arithmetic operations:

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

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

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

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

6.9 Binary bitwise operations

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

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

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

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

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

 $^{^2}$ If x is very close to an exact integer multiple of y, it's possible for x//y to be one larger than $(x-x^2y)$ //y due to rounding. In such cases, Python returns the latter result, in order to preserve that divmod(x,y) [0] * y + x % y be very close to x.

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

6.10 Comparisons

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

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

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

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

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

6.10.1 Value comparisons

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

Chapter *Objects, values and types* states that objects have a value (in addition to type and identity). The value of an object is a rather abstract notion in Python: For example, there is no canonical access method for an object's value. Also, there is no requirement that the value of an object should be constructed in a particular way, e.g. comprised of all its data attributes. Comparison operators implement a particular notion of what the value of an object is. One can think of them as defining the value of an object indirectly, by means of their comparison implementation.

Because all types are (direct or indirect) subtypes of object, they inherit the default comparison behavior from object. Types can customize their comparison behavior by implementing *rich comparison methods* like $___lt_{__}()$, described in *Basic customization*.

The default behavior for equality comparison (== and !=) is based on the identity of the objects. Hence, equality comparison of instances with the same identity results in equality, and equality comparison of instances with different identities results in inequality. A motivation for this default behavior is the desire that all objects should be reflexive (i.e. x is y implies x == y).

A default order comparison (<, >, <=,and >=) is not provided; an attempt raises TypeError. A motivation for this default behavior is the lack of a similar invariant as for equality.

The behavior of the default equality comparison, that instances with different identities are always unequal, may be in contrast to what types will need that have a sensible definition of object value and value-based equality. Such types will need to customize their comparison behavior, and in fact, a number of built-in types have done that.

The following list describes the comparison behavior of the most important built-in types.

• Numbers of built-in numeric types (typesnumeric) and of the standard library types fractions.Fraction and decimal.Decimal can be compared within and across their types, with the restriction that complex numbers do not support order comparison. Within the limits of the types involved, they compare mathematically (algorithmically) correct without loss of precision.

The not-a-number values float ('NaN') and decimal.Decimal('NaN') are special. Any ordered comparison of a number to a not-a-number value is false. A counter-intuitive implication is that not-a-number

6.10. Comparisons

values are not equal to themselves. For example, if x = float('NaN'), 3 < x, x < 3 and x == x are all false, while x != x is true. This behavior is compliant with IEEE 754.

- None and NotImplemented are singletons. **PEP 8** advises that comparisons for singletons should always be done with is or is not, never the equality operators.
- Binary sequences (instances of bytes or bytearray) can be compared within and across their types. They compare lexicographically using the numeric values of their elements.
- Strings (instances of str) compare lexicographically using the numerical Unicode code points (the result of the built-in function ord()) of their characters.³

Strings and binary sequences cannot be directly compared.

• Sequences (instances of tuple, list, or range) can be compared only within each of their types, with the restriction that ranges do not support order comparison. Equality comparison across these types results in inequality, and ordering comparison across these types raises TypeError.

Sequences compare lexicographically using comparison of corresponding elements. The built-in containers typically assume identical objects are equal to themselves. That lets them bypass equality tests for identical objects to improve performance and to maintain their internal invariants.

Lexicographical comparison between built-in collections works as follows:

- For two collections to compare equal, they must be of the same type, have the same length, and each pair of corresponding elements must compare equal (for example, [1,2] == (1,2) is false because the type is not the same).
- Collections that support order comparison are ordered the same as their first unequal elements (for example, [1,2,x] <= [1,2,y] has the same value as x <= y). If a corresponding element does not exist, the shorter collection is ordered first (for example, [1,2] < [1,2,3] is true).
- Mappings (instances of dict) compare equal if and only if they have equal (key, value) pairs. Equality comparison of the keys and values enforces reflexivity.

Order comparisons (<, >, <=, and >=) raise TypeError.

• Sets (instances of set or frozenset) can be compared within and across their types.

They define order comparison operators to mean subset and superset tests. Those relations do not define total orderings (for example, the two sets $\{1,2\}$ and $\{2,3\}$ are not equal, nor subsets of one another, nor supersets of one another). Accordingly, sets are not appropriate arguments for functions which depend on total ordering (for example, $\min()$, $\max()$, and sorted() produce undefined results given a list of sets as inputs).

Comparison of sets enforces reflexivity of its elements.

• Most other built-in types have no comparison methods implemented, so they inherit the default comparison behavior.

User-defined classes that customize their comparison behavior should follow some consistency rules, if possible:

• Equality comparison should be reflexive. In other words, identical objects should compare equal:

$$x$$
 is y implies $x == y$

• Comparison should be symmetric. In other words, the following expressions should have the same result:

$$x == y \text{ and } y == x$$

 $x != y \text{ and } y != x$

 $To compare strings \ at the level \ of \ abstract \ characters \ (that \ is, \ in \ a \ way \ intuitive \ to \ humans), \ use \ \verb"unicodedata.normalize" \ () \ .$

³ The Unicode standard distinguishes between *code points* (e.g. U+0041) and *abstract characters* (e.g. "LATIN CAPITAL LETTER A"). While most abstract characters in Unicode are only represented using one code point, there is a number of abstract characters that can in addition be represented using a sequence of more than one code point. For example, the abstract character "LATIN CAPITAL LETTER C WITH CEDILLA" can be represented as a single *precomposed character* at code position U+00C7, or as a sequence of a *base character* at code position U+0043 (LATIN CAPITAL LETTER C), followed by a *combining character* at code position U+0327 (COMBINING CEDILLA).

The comparison operators on strings compare at the level of Unicode code points. This may be counter-intuitive to humans. For example, "\u0007" == "\u0043\u00327" is False, even though both strings represent the same abstract character "LATIN CAPITAL LETTER C WITH CEDILLA".

```
x < y \text{ and } y > x

x <= y \text{ and } y >= x
```

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

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

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

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

```
x == y and not x != y

x < y and not x >= y (for total ordering)

x > y and not x <= y (for total ordering)
```

The last two expressions apply to totally ordered collections (e.g. to sequences, but not to sets or mappings). See also the total_ordering() decorator.

• The hash () result should be consistent with equality. Objects that are equal should either have the same hash value, or be marked as unhashable.

Python does not enforce these consistency rules. In fact, the not-a-number values are an example for not following these rules.

6.10.2 Membership test operations

The operators in and not in test for membership. x in s evaluates to True if x is a member of s, and False otherwise. x not in s returns the negation of x in s. All built-in sequences and set types support this as well as dictionary, for which in tests whether the dictionary has a given key. For container types such as list, tuple, set, frozenset, dict, or collections.deque, the expression x in y is equivalent to any (x is y or y == y == y for y == y == y == y for y == y ==

For the string and bytes types, x in y is True if and only if x is a substring of y. An equivalent test is y. find (x) != -1. Empty strings are always considered to be a substring of any other string, so "" in "abc" will return True.

For user-defined classes which define the __contains__() method, x in y returns True if y. __contains__(x) returns a true value, and False otherwise.

For user-defined classes which do not define $_contains_()$ but do define $_iter_()$, x in y is True if some value z, for which the expression x is z or x == z is true, is produced while iterating over y. If an exception is raised during the iteration, it is as if in raised that exception.

Lastly, the old-style iteration protocol is tried: if a class defines $_getitem_()$, x in y is True if and only if there is a non-negative integer index i such that x is y[i] or x == y[i], and no lower integer index raises the IndexError exception. (If any other exception is raised, it is as if in raised that exception).

The operator *not* in is defined to have the inverse truth value of in.

6.10.3 Identity comparisons

The operators is and is not test for an object's identity: x is y is true if and only if x and y are the same object. An Object's identity is determined using the id() function. x is not y yields the inverse truth value.

6.11 Boolean operations

```
or_test ::= and_test | or_test "or" and_test
and_test ::= not_test | and_test "and" not_test
not_test ::= comparison | "not" not_test
```

⁴ Due to automatic garbage-collection, free lists, and the dynamic nature of descriptors, you may notice seemingly unusual behaviour in certain uses of the *is* operator, like those involving comparisons between instance methods, or constants. Check their documentation for more info.

In the context of Boolean operations, and also when expressions are used by control flow statements, the following values are interpreted as false: False, None, numeric zero of all types, and empty strings and containers (including strings, tuples, lists, dictionaries, sets and frozensets). All other values are interpreted as true. User-defined objects can customize their truth value by providing a __bool__() method.

The operator *not* yields True if its argument is false, False otherwise.

The expression x and y first evaluates x; if x is false, its value is returned; otherwise, y is evaluated and the resulting value is returned.

The expression x or y first evaluates x; if x is true, its value is returned; otherwise, y is evaluated and the resulting value is returned.

Note that neither and nor or restrict the value and type they return to False and True, but rather return the last evaluated argument. This is sometimes useful, e.g., if s is a string that should be replaced by a default value if it is empty, the expression s or 'foo' yields the desired value. Because not has to create a new value, it returns a boolean value regardless of the type of its argument (for example, not 'foo' produces False rather than ''.)

6.12 Assignment expressions

```
assignment_expression ::= [identifier ":="] expression
```

An assignment expression (sometimes also called a "named expression" or "walrus") assigns an expression to an identifier, while also returning the value of the expression.

One common use case is when handling matched regular expressions:

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

Or, when processing a file stream in chunks:

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

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

Added in version 3.8: See PEP 572 for more details about assignment expressions.

6.13 Conditional expressions

Conditional expressions (sometimes called a "ternary operator") have the lowest priority of all Python operations.

The expression x if C else y first evaluates the condition, C rather than x. If C is true, x is evaluated and its value is returned; otherwise, y is evaluated and its value is returned.

See PEP 308 for more details about conditional expressions.

6.14 Lambdas

Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. The expression lambda parameters: expression yields a function object. The unnamed object behaves like a function object defined with:

```
def <lambda>(parameters):
    return expression
```

See section *Function definitions* for the syntax of parameter lists. Note that functions created with lambda expressions cannot contain statements or annotations.

6.15 Expression lists

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

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

Added in version 3.5: Iterable unpacking in expression lists, originally proposed by PEP 448.

Added in version 3.11: Any item in an expression list may be starred. See PEP 646.

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

6.16 Evaluation order

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

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

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

6.17 Operator precedence

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

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

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

⁵ The power operator ** binds less tightly than an arithmetic or bitwise unary operator on its right, that is, 2**-1 is 0.5. ⁶ The % operator is also used for string formatting; the same precedence applies.

SIMPLE STATEMENTS

A simple statement is comprised within a single logical line. Several simple statements may occur on a single line separated by semicolons. The syntax for simple statements is:

```
simple_stmt ::=
                   expression_stmt
                   | assert_stmt
                   | assignment_stmt
                   | augmented_assignment_stmt
                   | annotated_assignment_stmt
                   | pass_stmt
                   | del_stmt
                   | return_stmt
                   | yield_stmt
                   | raise_stmt
                   | break_stmt
                   | continue_stmt
                   | import_stmt
                   | future_stmt
                   | global_stmt
                   | nonlocal_stmt
                   | type_stmt
```

7.1 Expression statements

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

```
expression_stmt ::= starred_expression
```

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

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

7.2 Assignment statements

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

```
assignment_stmt ::= (target_list "=")+ (starred_expression | yield_expression)
```

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

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

Assignment is defined recursively depending on the form of the target (list). When a target is part of a mutable object (an attribute reference, subscription or slicing), the mutable object must ultimately perform the assignment and decide about its validity, and may raise an exception if the assignment is unacceptable. The rules observed by various types and the exceptions raised are given with the definition of the object types (see section *The standard type hierarchy*).

Assignment of an object to a target list, optionally enclosed in parentheses or square brackets, is recursively defined as follows.

- If the target list is a single target with no trailing comma, optionally in parentheses, the object is assigned to that target.
- Else:
 - If the target list contains one target prefixed with an asterisk, called a "starred" target: The object must be an iterable with at least as many items as there are targets in the target list, minus one. The first items of the iterable are assigned, from left to right, to the targets before the starred target. The final items of the iterable are assigned to the targets after the starred target. A list of the remaining items in the iterable is then assigned to the starred target (the list can be empty).
 - Else: The object must be an iterable with the same number of items as there are targets in the target list, and the items are assigned, from left to right, to the corresponding targets.

Assignment of an object to a single target is recursively defined as follows.

- If the target is an identifier (name):
 - If the name does not occur in a *global* or *nonlocal* statement in the current code block: the name is bound to the object in the current local namespace.
 - Otherwise: the name is bound to the object in the global namespace or the outer namespace determined by nonlocal, respectively.

The name is rebound if it was already bound. This may cause the reference count for the object previously bound to the name to reach zero, causing the object to be deallocated and its destructor (if it has one) to be called.

• If the target is an attribute reference: The primary expression in the reference is evaluated. It should yield an object with assignable attributes; if this is not the case, TypeError is raised. That object is then asked to assign the assigned object to the given attribute; if it cannot perform the assignment, it raises an exception (usually but not necessarily AttributeError).

Note: If the object is a class instance and the attribute reference occurs on both sides of the assignment operator, the right-hand side expression, a.x can access either an instance attribute or (if no instance attribute exists) a class attribute. The left-hand side target a.x is always set as an instance attribute, creating it if necessary. Thus, the two occurrences of a.x do not necessarily refer to the same attribute: if the right-hand side expression refers to a class attribute, the left-hand side creates a new instance attribute as the target of the assignment:

```
class Cls:

x = 3  # class variable (continues on next page)
```

(continued from previous page)

```
inst = Cls()
inst.x = inst.x + 1  # writes inst.x as 4 leaving Cls.x as 3
```

This description does not necessarily apply to descriptor attributes, such as properties created with property().

• If the target is a subscription: The primary expression in the reference is evaluated. It should yield either a mutable sequence object (such as a list) or a mapping object (such as a dictionary). Next, the subscript expression is evaluated.

If the primary is a mutable sequence object (such as a list), the subscript must yield an integer. If it is negative, the sequence's length is added to it. The resulting value must be a nonnegative integer less than the sequence's length, and the sequence is asked to assign the assigned object to its item with that index. If the index is out of range, IndexError is raised (assignment to a subscripted sequence cannot add new items to a list).

If the primary is a mapping object (such as a dictionary), the subscript must have a type compatible with the mapping's key type, and the mapping is then asked to create a key/value pair which maps the subscript to the assigned object. This can either replace an existing key/value pair with the same key value, or insert a new key/value pair (if no key with the same value existed).

For user-defined objects, the <u>__setitem__()</u> method is called with appropriate arguments.

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

CPython implementation detail: In the current implementation, the syntax for targets is taken to be the same as for expressions, and invalid syntax is rejected during the code generation phase, causing less detailed error messages.

Although the definition of assignment implies that overlaps between the left-hand side and the right-hand side are 'simultaneous' (for example a, b = b, a swaps two variables), overlaps *within* the collection of assigned-to variables occur left-to-right, sometimes resulting in confusion. For instance, the following program prints [0, 2]:

```
➢ See also
PEP 3132 - Extended Iterable Unpacking
The specification for the *target feature.
```

7.2.1 Augmented assignment statements

Augmented assignment is the combination, in a single statement, of a binary operation and an assignment statement:

(See section *Primaries* for the syntax definitions of the last three symbols.)

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

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

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

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

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

7.2.2 Annotated assignment statements

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

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

The assignment target is considered "simple" if it consists of a single name that is not enclosed in parentheses. For simple assignment targets, if in class or module scope, the annotations are evaluated and stored in a special class or module attribute __annotations__ that is a dictionary mapping from variable names (mangled if private) to evaluated annotations. This attribute is writable and is automatically created at the start of class or module body execution, if annotations are found statically.

If the assignment target is not simple (an attribute, subscript node, or parenthesized name), the annotation is evaluated if in class or module scope, but not stored.

If a name is annotated in a function scope, then this name is local for that scope. Annotations are never evaluated and stored in function scopes.

If the right hand side is present, an annotated assignment performs the actual assignment before evaluating annotations (where applicable). If the right hand side is not present for an expression target, then the interpreter evaluates the target except for the last __setitem_ () or __setattr_ () call.

See also

PEP 526 - Syntax for Variable Annotations

The proposal that added syntax for annotating the types of variables (including class variables and instance variables), instead of expressing them through comments.

PEP 484 - Type hints

The proposal that added the typing module to provide a standard syntax for type annotations that can be used in static analysis tools and IDEs.

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

7.3 The assert statement

Assert statements are a convenient way to insert debugging assertions into a program:

```
assert_stmt ∷= "assert" expression ["," expression]
```

The simple form, assert expression, is equivalent to

```
if __debug__:
    if not expression: raise AssertionError
```

The extended form, assert expression1, expression2, is equivalent to

```
if __debug__:
   if not expression1: raise AssertionError(expression2)
```

These equivalences assume that __debug__ and AssertionError refer to the built-in variables with those names. In the current implementation, the built-in variable __debug__ is True under normal circumstances, False when optimization is requested (command line option -O). The current code generator emits no code for an assert statement when optimization is requested at compile time. Note that it is unnecessary to include the source code for the expression that failed in the error message; it will be displayed as part of the stack trace.

Assignments to __debug__ are illegal. The value for the built-in variable is determined when the interpreter starts.

7.4 The pass statement

```
pass stmt ::= "pass"
```

pass is a null operation — when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically, but no code needs to be executed, for example:

```
def f(arg): pass # a function that does nothing (yet)
class C: pass # a class with no methods (yet)
```

7.5 The del statement

```
del_stmt ::= "del" target_list
```

Deletion is recursively defined very similar to the way assignment is defined. Rather than spelling it out in full details, here are some hints.

Deletion of a target list recursively deletes each target, from left to right.

Deletion of a name removes the binding of that name from the local or global namespace, depending on whether the name occurs in a <code>global</code> statement in the same code block. If the name is unbound, a <code>NameError</code> exception will be raised.

Deletion of attribute references, subscriptions and slicings is passed to the primary object involved; deletion of a slicing is in general equivalent to assignment of an empty slice of the right type (but even this is determined by the sliced object).

Changed in version 3.2: Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block.

7.6 The return statement

```
return_stmt ::= "return" [expression_list]
```

return may only occur syntactically nested in a function definition, not within a nested class definition.

If an expression list is present, it is evaluated, else None is substituted.

return leaves the current function call with the expression list (or None) as return value.

When return passes control out of a try statement with a finally clause, that finally clause is executed before really leaving the function.

In a generator function, the return statement indicates that the generator is done and will cause StopIteration to be raised. The returned value (if any) is used as an argument to construct StopIteration and becomes the StopIteration.value attribute.

In an asynchronous generator function, an empty return statement indicates that the asynchronous generator is done and will cause StopAsyncIteration to be raised. A non-empty return statement is a syntax error in an asynchronous generator function.

7.7 The yield statement

```
yield_stmt ::= yield_expression
```

A yield statement is semantically equivalent to a yield expression. The yield statement can be used to omit the parentheses that would otherwise be required in the equivalent yield expression statement. For example, the yield statements

```
yield <expr>
yield from <expr>
```

are equivalent to the yield expression statements

```
(yield <expr>)
(yield from <expr>)
```

Yield expressions and statements are only used when defining a *generator* function, and are only used in the body of the generator function. Using yield in a function definition is sufficient to cause that definition to create a generator function instead of a normal function.

For full details of yield semantics, refer to the Yield expressions section.

7.8 The raise statement

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

If no expressions are present, raise re-raises the exception that is currently being handled, which is also known as the *active exception*. If there isn't currently an active exception, a RuntimeError exception is raised indicating that this is an error.

Otherwise, raise evaluates the first expression as the exception object. It must be either a subclass or an instance of BaseException. If it is a class, the exception instance will be obtained when needed by instantiating the class with no arguments.

The *type* of the exception is the exception instance's class, the *value* is the instance itself.

A traceback object is normally created automatically when an exception is raised and attached to it as the __traceback__ attribute. You can create an exception and set your own traceback in one step using the

with_traceback() exception method (which returns the same exception instance, with its traceback set to its argument), like so:

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

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

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

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

```
>>> try:
... print(1 / 0)
... except:
... raise RuntimeError("Something bad happened") from None
...
(continues on next page)
```

(continued from previous page)

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

Additional information on exceptions can be found in section *Exceptions*, and information about handling exceptions is in section *The try statement*.

Changed in version 3.3: None is now permitted as Y in raise X from Y.

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

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

7.9 The break statement

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

break may only occur syntactically nested in a for or while loop, but not nested in a function or class definition within that loop.

It terminates the nearest enclosing loop, skipping the optional else clause if the loop has one.

If a for loop is terminated by break, the loop control target keeps its current value.

When break passes control out of a try statement with a finally clause, that finally clause is executed before really leaving the loop.

7.10 The continue statement

```
continue_stmt ∷= "continue"
```

continue may only occur syntactically nested in a for or while loop, but not nested in a function or class definition within that loop. It continues with the next cycle of the nearest enclosing loop.

When *continue* passes control out of a *try* statement with a *finally* clause, that finally clause is executed before really starting the next loop cycle.

7.11 The import statement

The basic import statement (no from clause) is executed in two steps:

- 1. find a module, loading and initializing it if necessary
- 2. define a name or names in the local namespace for the scope where the *import* statement occurs.

When the statement contains multiple clauses (separated by commas) the two steps are carried out separately for each clause, just as though the clauses had been separated out into individual import statements.

The details of the first step, finding and loading modules, are described in greater detail in the section on the *import system*, which also describes the various types of packages and modules that can be imported, as well as all the hooks that can be used to customize the import system. Note that failures in this step may indicate either that the module could not be located, *or* that an error occurred while initializing the module, which includes execution of the module's code.

If the requested module is retrieved successfully, it will be made available in the local namespace in one of three ways:

- If the module name is followed by as, then the name following as is bound directly to the imported module.
- If no other name is specified, and the module being imported is a top level module, the module's name is bound in the local namespace as a reference to the imported module
- If the module being imported is *not* a top level module, then the name of the top level package that contains the module is bound in the local namespace as a reference to the top level package. The imported module must be accessed using its full qualified name rather than directly

The from form uses a slightly more complex process:

- 1. find the module specified in the from clause, loading and initializing it if necessary;
- 2. for each of the identifiers specified in the *import* clauses:
 - 1. check if the imported module has an attribute by that name
 - 2. if not, attempt to import a submodule with that name and then check the imported module again for that attribute
 - 3. if the attribute is not found, ImportError is raised.
 - 4. otherwise, a reference to that value is stored in the local namespace, using the name in the as clause if it is present, otherwise using the attribute name

Examples:

If the list of identifiers is replaced by a star ('*'), all public names defined in the module are bound in the local namespace for the scope where the *import* statement occurs.

The *public names* defined by a module are determined by checking the module's namespace for a variable named __all__; if defined, it must be a sequence of strings which are names defined or imported by that module. The names given in __all__ are all considered public and are required to exist. If __all__ is not defined, the set of public names includes all names found in the module's namespace which do not begin with an underscore character ('_'). __all__ should contain the entire public API. It is intended to avoid accidentally exporting items that are not part of the API (such as library modules which were imported and used within the module).

The wild card form of import — from module import * — is only allowed at the module level. Attempting to use it in class or function definitions will raise a SyntaxError.

When specifying what module to import you do not have to specify the absolute name of the module. When a module or package is contained within another package it is possible to make a relative import within the same top package without having to mention the package name. By using leading dots in the specified module or package after <code>from</code> you can specify how high to traverse up the current package hierarchy without specifying exact names. One leading dot means the current package where the module making the import exists. Two dots means up one package level. Three dots is up two levels, etc. So if you execute <code>from . import mod</code> from a module in the <code>pkg</code> package then you will end up importing <code>pkg.mod</code>. If you execute <code>from ..subpkg2 import mod</code> from within <code>pkg.subpkg1</code> you

will import pkg.subpkg2.mod. The specification for relative imports is contained in the *Package Relative Imports* section.

importlib.import_module() is provided to support applications that determine dynamically the modules to be loaded

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

7.11.1 Future statements

A *future statement* is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python where the feature becomes standard.

The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard.

A future statement must appear near the top of the module. The only lines that can appear before a future statement are:

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

The only feature that requires using the future statement is annotations (see PEP 563).

All historical features enabled by the future statement are still recognized by Python 3. The list includes absolute_import, division, generators, generator_stop, unicode_literals, print_function, nested_scopes and with_statement. They are all redundant because they are always enabled, and only kept for backwards compatibility.

A future statement is recognized and treated specially at compile time: Changes to the semantics of core constructs are often implemented by generating different code. It may even be the case that a new feature introduces new incompatible syntax (such as a new reserved word), in which case the compiler may need to parse the module differently. Such decisions cannot be pushed off until runtime.

For any given release, the compiler knows which feature names have been defined, and raises a compile-time error if a future statement contains a feature not known to it.

The direct runtime semantics are the same as for any import statement: there is a standard module __future__, described later, and it will be imported in the usual way at the time the future statement is executed.

The interesting runtime semantics depend on the specific feature enabled by the future statement.

Note that there is nothing special about the statement:

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

That is not a future statement; it's an ordinary import statement with no special semantics or syntax restrictions.

Code compiled by calls to the built-in functions <code>exec()</code> and <code>compile()</code> that occur in a module <code>M</code> containing a future statement will, by default, use the new syntax or semantics associated with the future statement. This can be controlled by optional arguments to <code>compile()</code> — see the documentation of that function for details.

A future statement typed at an interactive interpreter prompt will take effect for the rest of the interpreter session. If an interpreter is started with the -i option, is passed a script name to execute, and the script includes a future statement, it will be in effect in the interactive session started after the script is executed.

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

7.12 The global statement

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

The <code>global</code> statement is a declaration which holds for the entire current code block. It means that the listed identifiers are to be interpreted as globals. It would be impossible to assign to a global variable without <code>global</code>, although free variables may refer to globals without being declared global.

Names listed in a <code>global</code> statement must not be used in the same code block textually preceding that <code>global</code> statement.

Names listed in a global statement must not be defined as formal parameters, or as targets in with statements or except clauses, or in a for target list, class definition, function definition, import statement, or variable annotation.

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

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

7.13 The nonlocal statement

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

When the definition of a function or class is nested (enclosed) within the definitions of other functions, its nonlocal scopes are the local scopes of the enclosing functions. The <code>nonlocal</code> statement causes the listed identifiers to refer to names previously bound in nonlocal scopes. It allows encapsulated code to rebind such nonlocal identifiers. If a name is bound in more than one nonlocal scope, the nearest binding is used. If a name is not bound in any nonlocal scope, or if there is no nonlocal scope, a <code>SyntaxError</code> is raised.

The nonlocal statement applies to the entire scope of a function or class body. A SyntaxError is raised if a variable is used or assigned to prior to its nonlocal declaration in the scope.

```
PEP 3104 - Access to Names in Outer Scopes
The specification for the nonlocal statement.
```

Programmer's note: nonlocal is a directive to the parser and applies only to code parsed along with it. See the note for the <code>qlobal</code> statement.

7.14 The type statement

```
type_stmt ::= 'type' identifier [type_params] "=" expression
```

The type statement declares a type alias, which is an instance of typing. TypeAliasType.

For example, the following statement creates a type alias:

```
type Point = tuple[float, float]
```

This code is roughly equivalent to:

```
annotation-def VALUE_OF_Point():
    return tuple[float, float]
Point = typing.TypeAliasType("Point", VALUE_OF_Point())
```

annotation-def indicates an *annotation scope*, which behaves mostly like a function, but with several small differences.

The value of the type alias is evaluated in the annotation scope. It is not evaluated when the type alias is created, but only when the value is accessed through the type alias's __value__ attribute (see *Lazy evaluation*). This allows the type alias to refer to names that are not yet defined.

Type aliases may be made generic by adding a *type parameter list* after the name. See *Generic type aliases* for more. type is a *soft keyword*.

Added in version 3.12.

See also

PEP 695 - Type Parameter Syntax

Introduced the type statement and syntax for generic classes and functions.

COMPOUND STATEMENTS

Compound statements contain (groups of) other statements; they affect or control the execution of those other statements in some way. In general, compound statements span multiple lines, although in simple incarnations a whole compound statement may be contained in one line.

The *if*, while and *for* statements implement traditional control flow constructs. *try* specifies exception handlers and/or cleanup code for a group of statements, while the with statement allows the execution of initialization and finalization code around a block of code. Function and class definitions are also syntactically compound statements.

A compound statement consists of one or more 'clauses.' A clause consists of a header and a 'suite.' The clause headers of a particular compound statement are all at the same indentation level. Each clause header begins with a uniquely identifying keyword and ends with a colon. A suite is a group of statements controlled by a clause. A suite can be one or more semicolon-separated simple statements on the same line as the header, following the header's colon, or it can be one or more indented statements on subsequent lines. Only the latter form of a suite can contain nested compound statements; the following is illegal, mostly because it wouldn't be clear to which if clause a following else clause would belong:

```
if test1: if test2: print(x)
```

Also note that the semicolon binds tighter than the colon in this context, so that in the following example, either all or none of the print () calls are executed:

```
if x < y < z: print(x); print(y); print(z)</pre>
```

Summarizing:

```
compound_stmt ::=
                     if_stmt
                     | while_stmt
                     | for_stmt
                     | try_stmt
                     | with_stmt
                     | match_stmt
                     | funcdef
                     | classdef
                     | async_with_stmt
                     | async_for_stmt
                     | async_funcdef
suite
                     stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
                ::=
statement
                ::=
                     stmt_list NEWLINE | compound_stmt
stmt list
                ::=
                     simple_stmt (";" simple_stmt)* [";"]
```

Note that statements always end in a NEWLINE possibly followed by a DEDENT. Also note that optional continuation clauses always begin with a keyword that cannot start a statement, thus there are no ambiguities (the 'dangling else' problem is solved in Python by requiring nested if statements to be indented).

The formatting of the grammar rules in the following sections places each clause on a separate line for clarity.

8.1 The if statement

The *if* statement is used for conditional execution:

It selects exactly one of the suites by evaluating the expressions one by one until one is found to be true (see section *Boolean operations* for the definition of true and false); then that suite is executed (and no other part of the *if* statement is executed or evaluated). If all expressions are false, the suite of the *else* clause, if present, is executed.

8.2 The while statement

The while statement is used for repeated execution as long as an expression is true:

```
while_stmt ::= "while" assignment_expression ":" suite
["else" ":" suite]
```

This repeatedly tests the expression and, if it is true, executes the first suite; if the expression is false (which may be the first time it is tested) the suite of the else clause, if present, is executed and the loop terminates.

A break statement executed in the first suite terminates the loop without executing the else clause's suite. A continue statement executed in the first suite skips the rest of the suite and goes back to testing the expression.

8.3 The for statement

The for statement is used to iterate over the elements of a sequence (such as a string, tuple or list) or other iterable object:

The starred_list expression is evaluated once; it should yield an *iterable* object. An *iterator* is created for that iterable. The first item provided by the iterator is then assigned to the target list using the standard rules for assignments (see *Assignment statements*), and the suite is executed. This repeats for each item provided by the iterator. When the iterator is exhausted, the suite in the else clause, if present, is executed, and the loop terminates.

A break statement executed in the first suite terminates the loop without executing the else clause's suite. A continue statement executed in the first suite skips the rest of the suite and continues with the next item, or with the else clause if there is no next item.

The for-loop makes assignments to the variables in the target list. This overwrites all previous assignments to those variables including those made in the suite of the for-loop:

Names in the target list are not deleted when the loop is finished, but if the sequence is empty, they will not have been assigned to at all by the loop. Hint: the built-in type range () represents immutable arithmetic sequences of integers. For instance, iterating range (3) successively yields 0, 1, and then 2.

Changed in version 3.11: Starred elements are now allowed in the expression list.

8.4 The try statement

The try statement specifies exception handlers and/or cleanup code for a group of statements:

```
try_stmt
                 try1_stmt | try2_stmt | try3_stmt
           ::=
                 "try" ":" suite
try1_stmt
          ::=
                 ("except" [expression ["as" identifier]] ":" suite)+
                 ["else" ":" suite]
                 ["finally" ":" suite]
                 "try" ":" suite
try2_stmt ::=
                 ("except" "*" expression ["as" identifier] ":" suite) +
                 ["else" ":" suite]
                 ["finally" ":" suite]
                 "try" ":" suite
          ::=
try3_stmt
                 "finally" ": " suite
```

Additional information on exceptions can be found in section *Exceptions*, and information on using the *raise* statement to generate exceptions may be found in section *The raise statement*.

8.4.1 except clause

The except clause(s) specify one or more exception handlers. When no exception occurs in the try clause, no exception handler is executed. When an exception occurs in the try suite, a search for an exception handler is started. This search inspects the except clauses in turn until one is found that matches the exception. An expression-less except clause, if present, must be last; it matches any exception.

For an except clause with an expression, the expression must evaluate to an exception type or a tuple of exception types. The raised exception matches an except clause whose expression evaluates to the class or a *non-virtual base class* of the exception object, or to a tuple that contains such a class.

If no except clause matches the exception, the search for an exception handler continues in the surrounding code and on the invocation stack.¹

If the evaluation of an expression in the header of an except clause raises an exception, the original search for a handler is canceled and a search starts for the new exception in the surrounding code and on the call stack (it is treated as if the entire try statement raised the exception).

When a matching except clause is found, the exception is assigned to the target specified after the as keyword in that except clause, if present, and the except clause's suite is executed. All except clauses must have an executable block. When the end of this block is reached, execution continues normally after the entire try statement. (This means that if two nested handlers exist for the same exception, and the exception occurs in the try clause of the inner handler, the outer handler will not handle the exception.)

When an exception has been assigned using as target, it is cleared at the end of the except clause. This is as if

```
except E as N:
    foo
```

was translated to

```
except E as N:
    try:
        foo
    finally:
        del N
```

¹ The exception is propagated to the invocation stack unless there is a *finally* clause which happens to raise another exception. That new exception causes the old one to be lost.

This means the exception must be assigned to a different name to be able to refer to it after the except clause. Exceptions are cleared because with the traceback attached to them, they form a reference cycle with the stack frame, keeping all locals in that frame alive until the next garbage collection occurs.

Before an except clause's suite is executed, the exception is stored in the sys module, where it can be accessed from within the body of the except clause by calling sys.exception(). When leaving an exception handler, the exception stored in the sys module is reset to its previous value:

```
>>> print(sys.exception())
None
>>> try:
       raise TypeError
... except:
      print(repr(sys.exception()))
            raise ValueError
       except:
           print(repr(sys.exception()))
        print(repr(sys.exception()))
. . .
TypeError()
ValueError()
TypeError()
>>> print(sys.exception())
None
```

8.4.2 except * clause

The except* clause(s) are used for handling ExceptionGroups. The exception type for matching is interpreted as in the case of except, but in the case of exception groups we can have partial matches when the type matches some of the exceptions in the group. This means that multiple except* clauses can execute, each handling part of the exception group. Each clause executes at most once and handles an exception group of all matching exceptions. Each exception in the group is handled by at most one except* clause, the first that matches it.

Any remaining exceptions that were not handled by any <code>except*</code> clause are re-raised at the end, along with all exceptions that were raised from within the <code>except*</code> clauses. If this list contains more than one exception to reraise, they are combined into an exception group.

If the raised exception is not an exception group and its type matches one of the except* clauses, it is caught and wrapped by an exception group with an empty message string.

```
>>> try:
... raise BlockingIOError
... except* BlockingIOError as e:
... print(repr(e))
...
ExceptionGroup('', (BlockingIOError()))
```

An except* clause must have a matching expression; it cannot be except*:. Furthermore, this expression cannot contain exception group types, because that would have ambiguous semantics.

It is not possible to mix except and except* in the same try. break, continue and return cannot appear in an except* clause.

8.4.3 else clause

The optional else clause is executed if the control flow leaves the *try* suite, no exception was raised, and no *return*, *continue*, or *break* statement was executed. Exceptions in the else clause are not handled by the preceding *except* clauses.

8.4.4 finally clause

If finally is present, it specifies a 'cleanup' handler. The try clause is executed, including any except and else clauses. If an exception occurs in any of the clauses and is not handled, the exception is temporarily saved. The finally clause is executed. If there is a saved exception it is re-raised at the end of the finally clause. If the finally clause raises another exception, the saved exception is set as the context of the new exception. If the finally clause executes a return, break or continue statement, the saved exception is discarded:

```
>>> def f():
... try:
... 1/0
... finally:
... return 42
...
>>> f()
42
```

The exception information is not available to the program during execution of the finally clause.

When a return, break or continue statement is executed in the try suite of a try...finally statement, the finally clause is also executed 'on the way out.'

The return value of a function is determined by the last return statement executed. Since the finally clause always executes, a return statement executed in the finally clause will always be the last one executed:

Changed in version 3.8: Prior to Python 3.8, a *continue* statement was illegal in the finally clause due to a problem with the implementation.

8.5 The with statement

The with statement is used to wrap the execution of a block with methods defined by a context manager (see section With Statement Context Managers). This allows common try...except...finally usage patterns to be encapsulated for convenient reuse.

```
with_stmt ::= "with" ( "(" with_stmt_contents ","? ")" | with_stmt_contents ) ":" sum
with_stmt_contents ::= with_item ("," with_item)*
with_item ::= expression ["as" target]
```

The execution of the with statement with one "item" proceeds as follows:

- 1. The context expression (the expression given in the with_item) is evaluated to obtain a context manager.
- 2. The context manager's __enter__() is loaded for later use.
- 3. The context manager's __exit__() is loaded for later use.
- 4. The context manager's __enter__() method is invoked.
- 5. If a target was included in the with statement, the return value from __enter__() is assigned to it.

1 Note

The with statement guarantees that if the __enter__() method returns without an error, then __exit__() will always be called. Thus, if an error occurs during the assignment to the target list, it will be treated the same as an error occurring within the suite would be. See step 7 below.

- 6. The suite is executed.
- 7. The context manager's __exit__() method is invoked. If an exception caused the suite to be exited, its type, value, and traceback are passed as arguments to __exit__(). Otherwise, three None arguments are supplied.

If the suite was exited due to an exception, and the return value from the $__exit__()$ method was false, the exception is reraised. If the return value was true, the exception is suppressed, and execution continues with the statement following the with statement.

If the suite was exited for any reason other than an exception, the return value from __exit__() is ignored, and execution proceeds at the normal location for the kind of exit that was taken.

The following code:

```
with EXPRESSION as TARGET:
SUITE
```

is semantically equivalent to:

```
manager = (EXPRESSION)
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager)

try:
    TARGET = value
    SUITE
except:
    if not exit(manager, *sys.exc_info()):
        raise
else:
    exit(manager, None, None)
```

With more than one item, the context managers are processed as if multiple with statements were nested:

```
with A() as a, B() as b:
SUITE
```

is semantically equivalent to:

```
with A() as a:
    with B() as b:
    SUITE
```

You can also write multi-item context managers in multiple lines if the items are surrounded by parentheses. For example:

```
with (
    A() as a,
    B() as b,
):
    SUITE
```

Changed in version 3.1: Support for multiple context expressions.

Changed in version 3.10: Support for using grouping parentheses to break the statement in multiple lines.

```
PEP 343 - The "with" statement
The specification, background, and examples for the Python with statement.
```

8.6 The match statement

Added in version 3.10.

The match statement is used for pattern matching. Syntax:

1 Note

This section uses single quotes to denote *soft keywords*.

Pattern matching takes a pattern as input (following case) and a subject value (following match). The pattern (which may contain subpatterns) is matched against the subject value. The outcomes are:

- A match success or failure (also termed a pattern success or failure).
- Possible binding of matched values to a name. The prerequisites for this are further discussed below.

The match and case keywords are soft keywords.

```
    ◆ See also
    • PEP 634 – Structural Pattern Matching: Specification
```

• PEP 636 - Structural Pattern Matching: Tutorial

8.6.1 Overview

Here's an overview of the logical flow of a match statement:

- 1. The subject expression subject_expr is evaluated and a resulting subject value obtained. If the subject expression contains a comma, a tuple is constructed using the standard rules.
- 2. Each pattern in a case_block is attempted to match with the subject value. The specific rules for success or failure are described below. The match attempt can also bind some or all of the standalone names within the pattern. The precise pattern binding rules vary per pattern type and are specified below. Name bindings made during a successful pattern match outlive the executed block and can be used after the match statement.

1 Note

During failed pattern matches, some subpatterns may succeed. Do not rely on bindings being made for a failed match. Conversely, do not rely on variables remaining unchanged after a failed match. The exact behavior is dependent on implementation and may vary. This is an intentional decision made to allow different implementations to add optimizations.

- 3. If the pattern succeeds, the corresponding guard (if present) is evaluated. In this case all name bindings are guaranteed to have happened.
 - If the guard evaluates as true or is missing, the block inside case_block is executed.
 - Otherwise, the next case_block is attempted as described above.
 - If there are no further case blocks, the match statement is completed.

1 Note

Users should generally never rely on a pattern being evaluated. Depending on implementation, the interpreter may cache values or use other optimizations which skip repeated evaluations.

A sample match statement:

In this case, if flag is a guard. Read more about that in the next section.

8.6.2 Guards

```
quard ::= "if" named_expression
```

A guard (which is part of the case) must succeed for code inside the case block to execute. It takes the form: if followed by an expression.

The logical flow of a case block with a guard follows:

- 1. Check that the pattern in the case block succeeded. If the pattern failed, the guard is not evaluated and the next case block is checked.
- 2. If the pattern succeeded, evaluate the guard.
 - If the guard condition evaluates as true, the case block is selected.
 - If the guard condition evaluates as false, the case block is not selected.
 - If the quard raises an exception during evaluation, the exception bubbles up.

Guards are allowed to have side effects as they are expressions. Guard evaluation must proceed from the first to the last case block, one at a time, skipping case blocks whose pattern(s) don't all succeed. (I.e., guard evaluation must happen in order.) Guard evaluation must stop once a case block is selected.

8.6.3 Irrefutable Case Blocks

An irrefutable case block is a match-all case block. A match statement may have at most one irrefutable case block, and it must be last.

A case block is considered irrefutable if it has no guard and its pattern is irrefutable. A pattern is considered irrefutable if we can prove from its syntax alone that it will always succeed. Only the following patterns are irrefutable:

- AS Patterns whose left-hand side is irrefutable
- OR Patterns containing at least one irrefutable pattern
- Capture Patterns
- Wildcard Patterns
- parenthesized irrefutable patterns

8.6.4 Patterns

1 Note

This section uses grammar notations beyond standard EBNF:

- the notation SEP.RULE+ is shorthand for RULE (SEP RULE) *
- the notation ! RULE is shorthand for a negative lookahead assertion

The top-level syntax for patterns is:

```
| class_pattern
```

The descriptions below will include a description "in simple terms" of what a pattern does for illustration purposes (credits to Raymond Hettinger for a document that inspired most of the descriptions). Note that these descriptions are purely for illustration purposes and **may not** reflect the underlying implementation. Furthermore, they do not cover all valid forms.

OR Patterns

An OR pattern is two or more patterns separated by vertical bars |. Syntax:

```
or_pattern ::= "|".closed_pattern+
```

Only the final subpattern may be *irrefutable*, and each subpattern must bind the same set of names to avoid ambiguity.

An OR pattern matches each of its subpatterns in turn to the subject value, until one succeeds. The OR pattern is then considered successful. Otherwise, if none of the subpatterns succeed, the OR pattern fails.

In simple terms, $P1 + P2 + \dots$ will try to match P1, if it fails it will try to match P2, succeeding immediately if any succeeds, failing otherwise.

AS Patterns

An AS pattern matches an OR pattern on the left of the as keyword against a subject. Syntax:

```
as_pattern ::= or_pattern "as" capture_pattern
```

If the OR pattern fails, the AS pattern fails. Otherwise, the AS pattern binds the subject to the name on the right of the as keyword and succeeds. capture_pattern cannot be a _.

In simple terms P as NAME will match with P, and on success it will set NAME = <subject>.

Literal Patterns

A literal pattern corresponds to most *literals* in Python. Syntax:

The rule strings and the token NUMBER are defined in the *standard Python grammar*. Triple-quoted strings are supported. Raw strings and byte strings are supported. *f-strings* are not supported.

The forms signed_number '+' NUMBER and signed_number '-' NUMBER are for expressing *complex numbers*; they require a real number on the left and an imaginary number on the right. E.g. 3 + 4j.

In simple terms, LITERAL will succeed only if <subject> == LITERAL. For the singletons None, True and False, the *is* operator is used.

Capture Patterns

A capture pattern binds the subject value to a name. Syntax:

```
capture_pattern ::= !'_' NAME
```

A single underscore _ is not a capture pattern (this is what !'_' expresses). It is instead treated as a wildcard_pattern.

In a given pattern, a given name can only be bound once. E.g. case x, x: ... is invalid while case [x] | x: . . . is allowed.

Capture patterns always succeed. The binding follows scoping rules established by the assignment expression operator in PEP 572; the name becomes a local variable in the closest containing function scope unless there's an applicable global or nonlocal statement.

In simple terms NAME will always succeed and it will set NAME = <subject>.

Wildcard Patterns

A wildcard pattern always succeeds (matches anything) and binds no name. Syntax:

```
wildcard_pattern ::=
```

_ is a soft keyword within any pattern, but only within patterns. It is an identifier, as usual, even within match subject expressions, guards, and case blocks.

In simple terms, _ will always succeed.

Value Patterns

A value pattern represents a named value in Python. Syntax:

```
value_pattern ::= attr
              ::= name_or_attr "." NAME
attr
name_or_attr ::= attr | NAME
```

The dotted name in the pattern is looked up using standard Python name resolution rules. The pattern succeeds if the value found compares equal to the subject value (using the == equality operator).

In simple terms NAME1.NAME2 will succeed only if <subject> == NAME1.NAME2



1 Note

If the same value occurs multiple times in the same match statement, the interpreter may cache the first value found and reuse it rather than repeat the same lookup. This cache is strictly tied to a given execution of a given match statement.

Group Patterns

A group pattern allows users to add parentheses around patterns to emphasize the intended grouping. Otherwise, it has no additional syntax. Syntax:

```
"(" pattern ")"
group_pattern ::=
```

In simple terms (P) has the same effect as P.

Sequence Patterns

A sequence pattern contains several subpatterns to be matched against sequence elements. The syntax is similar to the unpacking of a list or tuple.

```
"[" [maybe_sequence_pattern] "]"
sequence_pattern
                       ::=
                             | "(" [open_sequence_pattern] ")"
                            maybe_star_pattern "," [maybe_sequence_pattern]
open_sequence_pattern
                       ::=
maybe_sequence_pattern ::=
                             ",".maybe_star_pattern+ ","?
maybe_star_pattern
                       ::=
                             star_pattern | pattern
star_pattern
                        ::=
                             "*" (capture_pattern | wildcard_pattern)
```

There is no difference if parentheses or square brackets are used for sequence patterns (i.e. (...) vs [...]).

1 Note

A single pattern enclosed in parentheses without a trailing comma (e.g. (3 | 4)) is a group pattern. While a single pattern enclosed in square brackets (e.g. [3 | 4]) is still a sequence pattern.

At most one star subpattern may be in a sequence pattern. The star subpattern may occur in any position. If no star subpattern is present, the sequence pattern is a fixed-length sequence pattern; otherwise it is a variable-length sequence pattern.

The following is the logical flow for matching a sequence pattern against a subject value:

- 1. If the subject value is not a sequence², the sequence pattern fails.
- 2. If the subject value is an instance of str, bytes or bytearray the sequence pattern fails.
- 3. The subsequent steps depend on whether the sequence pattern is fixed or variable-length.

If the sequence pattern is fixed-length:

- 1. If the length of the subject sequence is not equal to the number of subpatterns, the sequence pattern fails
- 2. Subpatterns in the sequence pattern are matched to their corresponding items in the subject sequence from left to right. Matching stops as soon as a subpattern fails. If all subpatterns succeed in matching their corresponding item, the sequence pattern succeeds.

Otherwise, if the sequence pattern is variable-length:

- 1. If the length of the subject sequence is less than the number of non-star subpatterns, the sequence pattern
- 2. The leading non-star subpatterns are matched to their corresponding items as for fixed-length sequences.
- ² In pattern matching, a sequence is defined as one of the following:
- a class that inherits from collections.abc.Sequence
- a Python class that has been registered as collections.abc.Sequence
- a builtin class that has its (CPython) Py_TPFLAGS_SEQUENCE bit set
- · a class that inherits from any of the above

The following standard library classes are sequences:

- array.array
- · collections.deque
- list
- memoryview
- range
- tuple



Subject values of type str, bytes, and bytearray do not match sequence patterns.

- 3. If the previous step succeeds, the star subpattern matches a list formed of the remaining subject items, excluding the remaining items corresponding to non-star subpatterns following the star subpattern.
- 4. Remaining non-star subpatterns are matched to their corresponding subject items, as for a fixed-length sequence.



The length of the subject sequence is obtained via len() (i.e. via the __len__() protocol). This length may be cached by the interpreter in a similar manner as *value patterns*.

In simple terms [P1, P2, P3, ..., P<N>] matches only if all the following happens:

- check <subject> is a sequence
- len(subject) == <N>
- P1 matches <subject>[0] (note that this match can also bind names)
- P2 matches <subject>[1] (note that this match can also bind names)
- ... and so on for the corresponding pattern/element.

Mapping Patterns

A mapping pattern contains one or more key-value patterns. The syntax is similar to the construction of a dictionary. Syntax:

```
mapping_pattern ::= "{" [items_pattern] "}"
items_pattern ::= ",".key_value_pattern+ ","?
key_value_pattern ::= (literal_pattern | value_pattern) ":" pattern
| double_star_pattern
double_star_pattern ::= "**" capture_pattern
```

At most one double star pattern may be in a mapping pattern. The double star pattern must be the last subpattern in the mapping pattern.

Duplicate keys in mapping patterns are disallowed. Duplicate literal keys will raise a SyntaxError. Two keys that otherwise have the same value will raise a ValueError at runtime.

The following is the logical flow for matching a mapping pattern against a subject value:

- 1. If the subject value is not a mapping³, the mapping pattern fails.
- 2. If every key given in the mapping pattern is present in the subject mapping, and the pattern for each key matches the corresponding item of the subject mapping, the mapping pattern succeeds.
- 3. If duplicate keys are detected in the mapping pattern, the pattern is considered invalid. A SyntaxError is raised for duplicate literal values; or a ValueError for named keys of the same value.



Key-value pairs are matched using the two-argument form of the mapping subject's <code>get()</code> method. Matched key-value pairs must already be present in the mapping, and not created on-the-fly via <code>__missing__()</code> or <code>__getitem__()</code>.

The standard library classes dict and types. MappingProxyType are mappings.

³ In pattern matching, a mapping is defined as one of the following:

⁻ a class that inherits from ${\tt collections.abc.Mapping}$

[•] a Python class that has been registered as collections.abc.Mapping

[•] a builtin class that has its (CPython) Py_TPFLAGS_MAPPING bit set

[•] a class that inherits from any of the above

In simple terms {KEY1: P1, KEY2: P2, ... } matches only if all the following happens:

- check <subject> is a mapping
- KEY1 in <subject>
- P1 matches <subject>[KEY1]
- ... and so on for the corresponding KEY/pattern pair.

Class Patterns

A class pattern represents a class and its positional and keyword arguments (if any). Syntax:

The same keyword should not be repeated in class patterns.

The following is the logical flow for matching a class pattern against a subject value:

- 1. If name_or_attr is not an instance of the builtin type, raise TypeError.
- 2. If the subject value is not an instance of name_or_attr (tested via isinstance ()), the class pattern fails.
- 3. If no pattern arguments are present, the pattern succeeds. Otherwise, the subsequent steps depend on whether keyword or positional argument patterns are present.

For a number of built-in types (specified below), a single positional subpattern is accepted which will match the entire subject; for these types keyword patterns also work as for other types.

If only keyword patterns are present, they are processed as follows, one by one:

- I. The keyword is looked up as an attribute on the subject.
 - If this raises an exception other than AttributeError, the exception bubbles up.
 - If this raises AttributeError, the class pattern has failed.
 - Else, the subpattern associated with the keyword pattern is matched against the subject's attribute value. If this fails, the class pattern fails; if this succeeds, the match proceeds to the next keyword.
- II. If all keyword patterns succeed, the class pattern succeeds.

If any positional patterns are present, they are converted to keyword patterns using the <u>__match_args__</u> attribute on the class name_or_attr before matching:

- I. The equivalent of getattr(cls, "__match_args__", ()) is called.
 - If this raises an exception, the exception bubbles up.
 - If the returned value is not a tuple, the conversion fails and TypeError is raised.
 - If there are more positional patterns than len(cls.__match_args__), TypeError is raised.
 - Otherwise, positional pattern i is converted to a keyword pattern using __match_args__[i] as the keyword. __match_args__[i] must be a string; if not TypeError is raised.
 - If there are duplicate keywords, TypeError is raised.

See also

Customizing positional arguments in class pattern matching

II. Once all positional patterns have been converted to keyword patterns,

the match proceeds as if there were only keyword patterns.

For the following built-in types the handling of positional subpatterns is different:

- bool
- bytearray
- bytes
- dict
- float
- frozenset
- int
- list
- set
- str
- tuple

These classes accept a single positional argument, and the pattern there is matched against the whole object rather than an attribute. For example int (0|1) matches the value 0, but not the value 0.0.

In simple terms CLS (P1, attr=P2) matches only if the following happens:

- isinstance(<subject>, CLS)
- convert P1 to a keyword pattern using CLS.__match_args__
- For each keyword argument attr=P2:

```
- hasattr(<subject>, "attr")
```

- P2 matches <subject>.attr
- ... and so on for the corresponding keyword argument/pattern pair.

See also

- PEP 634 Structural Pattern Matching: Specification
- PEP 636 Structural Pattern Matching: Tutorial

8.7 Function definitions

A function definition defines a user-defined function object (see section *The standard type hierarchy*):

```
defparameter ("," defparameter)* ["," [parameter_list_starargs]]
parameter_list_no_posonly ::=
                                 | parameter_list_starargs
                                 "*" [star_parameter] ("," defparameter) * ["," ["**" parameter ['
parameter_list_starargs
                                 | "**" parameter [","]
parameter
                           ::=
                                 identifier [":" expression]
                                 identifier [":" ["*"] expression]
                           ::=
star_parameter
                                 parameter ["=" expression]
defparameter
                           ::=
funcname
                                 identifier
```

A function definition is an executable statement. Its execution binds the function name in the current local namespace to a function object (a wrapper around the executable code for the function). This function object contains a reference to the current global namespace as the global namespace to be used when the function is called.

The function definition does not execute the function body; this gets executed only when the function is called.⁴

A function definition may be wrapped by one or more *decorator* expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition. The result must be a callable, which is invoked with the function object as the only argument. The returned value is bound to the function name instead of the function object. Multiple decorators are applied in nested fashion. For example, the following code

```
@f1(arg)
@f2
def func(): pass
```

is roughly equivalent to

```
def func(): pass
func = f1(arg)(f2(func))
```

except that the original function is not temporarily bound to the name func.

Changed in version 3.9: Functions may be decorated with any valid assignment_expression. Previously, the grammar was much more restrictive; see PEP 614 for details.

A list of *type parameters* may be given in square brackets between the function's name and the opening parenthesis for its parameter list. This indicates to static type checkers that the function is generic. At runtime, the type parameters can be retrieved from the function's __type_params__ attribute. See *Generic functions* for more.

Changed in version 3.12: Type parameter lists are new in Python 3.12.

When one or more *parameters* have the form *parameter* = *expression*, the function is said to have "default parameter values." For a parameter with a default value, the corresponding *argument* may be omitted from a call, in which case the parameter's default value is substituted. If a parameter has a default value, all following parameters up until the "*" must also have a default value — this is a syntactic restriction that is not expressed by the grammar.

Default parameter values are evaluated from left to right when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same "pre-computed" value is used for each call. This is especially important to understand when a default parameter value is a mutable object, such as a list or a dictionary: if the function modifies the object (e.g. by appending an item to a list), the default parameter value is in effect modified. This is generally not what was intended. A way around this is to use None as the default, and explicitly test for it in the body of the function, e.g.:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Function call semantics are described in more detail in section *Calls*. A function call always assigns values to all parameters mentioned in the parameter list, either from positional arguments, from keyword arguments, or from default

⁴ A string literal appearing as the first statement in the function body is transformed into the function's <u>__doc__</u> attribute and therefore the function at the functi

values. If the form "*identifier" is present, it is initialized to a tuple receiving any excess positional parameters, defaulting to the empty tuple. If the form "**identifier" is present, it is initialized to a new ordered mapping receiving any excess keyword arguments, defaulting to a new empty mapping of the same type. Parameters after "*" or "*identifier" are keyword-only parameters and may only be passed by keyword arguments. Parameters before "/" are positional-only parameters and may only be passed by positional arguments.

Changed in version 3.8: The / function parameter syntax may be used to indicate positional-only parameters. See **PEP 570** for details.

Parameters may have an *annotation* of the form ": expression" following the parameter name. Any parameter may have an annotation, even those of the form *identifier or **identifier. (As a special case, parameters of the form *identifier may have an annotation ": *expression".) Functions may have "return" annotation of the form "-> expression" after the parameter list. These annotations can be any valid Python expression. The presence of annotations does not change the semantics of a function. The annotation values are available as values of a dictionary keyed by the parameters' names in the __annotations__ attribute of the function object. If the annotations import from __future__ is used, annotations are preserved as strings at runtime which enables postponed evaluation. Otherwise, they are evaluated when the function definition is executed. In this case annotations may be evaluated in a different order than they appear in the source code.

Changed in version 3.11: Parameters of the form "*identifier" may have an annotation ": *expression". See PEP 646.

It is also possible to create anonymous functions (functions not bound to a name), for immediate use in expressions. This uses lambda expressions, described in section *Lambdas*. Note that the lambda expression is merely a shorthand for a simplified function definition; a function defined in a "def" statement can be passed around or assigned to another name just like a function defined by a lambda expression. The "def" form is actually more powerful since it allows the execution of multiple statements and annotations.

Programmer's note: Functions are first-class objects. A "def" statement executed inside a function definition defines a local function that can be returned or passed around. Free variables used in the nested function can access the local variables of the function containing the def. See section *Naming and binding* for details.

See also

PEP 3107 - Function Annotations

The original specification for function annotations.

PEP 484 - Type Hints

Definition of a standard meaning for annotations: type hints.

PEP 526 - Syntax for Variable Annotations

Ability to type hint variable declarations, including class variables and instance variables.

PEP 563 - Postponed Evaluation of Annotations

Support for forward references within annotations by preserving annotations in a string form at runtime instead of eager evaluation.

PEP 318 - Decorators for Functions and Methods

Function and method decorators were introduced. Class decorators were introduced in PEP 3129.

8.8 Class definitions

A class definition defines a class object (see section The standard type hierarchy):

```
classdef ::= [decorators] "class" classname [type_params] [inheritance] ":" suite inheritance ::= "(" [argument_list] ")" classname ::= identifier
```

A class definition is an executable statement. The inheritance list usually gives a list of base classes (see *Metaclasses* for more advanced uses), so each item in the list should evaluate to a class object which allows subclassing. Classes

8.8. Class definitions 125

without an inheritance list inherit, by default, from the base class object; hence,

```
class Foo:
   pass
```

is equivalent to

```
class Foo(object):
   pass
```

The class's suite is then executed in a new execution frame (see *Naming and binding*), using a newly created local namespace and the original global namespace. (Usually, the suite contains mostly function definitions.) When the class's suite finishes execution, its execution frame is discarded but its local namespace is saved.⁵ A class object is then created using the inheritance list for the base classes and the saved local namespace for the attribute dictionary. The class name is bound to this class object in the original local namespace.

The order in which attributes are defined in the class body is preserved in the new class's __dict__. Note that this is reliable only right after the class is created and only for classes that were defined using the definition syntax.

Class creation can be customized heavily using *metaclasses*.

Classes can also be decorated: just like when decorating functions,

```
@f1 (arg)
@f2
class Foo: pass
```

is roughly equivalent to

```
class Foo: pass
Foo = f1(arg)(f2(Foo))
```

The evaluation rules for the decorator expressions are the same as for function decorators. The result is then bound to the class name.

Changed in version 3.9: Classes may be decorated with any valid assignment_expression. Previously, the grammar was much more restrictive; see PEP 614 for details.

A list of *type parameters* may be given in square brackets immediately after the class's name. This indicates to static type checkers that the class is generic. At runtime, the type parameters can be retrieved from the class's __type_params__ attribute. See *Generic classes* for more.

Changed in version 3.12: Type parameter lists are new in Python 3.12.

Programmer's note: Variables defined in the class definition are class attributes; they are shared by instances. Instance attributes can be set in a method with self.name = value. Both class and instance attributes are accessible through the notation "self.name", and an instance attribute hides a class attribute with the same name when accessed in this way. Class attributes can be used as defaults for instance attributes, but using mutable values there can lead to unexpected results. *Descriptors* can be used to create instance variables with different implementation details.

```
See also
```

PEP 3115 - Metaclasses in Python 3000

The proposal that changed the declaration of metaclasses to the current syntax, and the semantics for how classes with metaclasses are constructed.

PEP 3129 - Class Decorators

The proposal that added class decorators. Function and method decorators were introduced in PEP 318.

⁵ A string literal appearing as the first statement in the class body is transformed into the namespace's __doc__ item and therefore the class's docstring.

8.9 Coroutines

Added in version 3.5.

8.9.1 Coroutine function definition

Execution of Python coroutines can be suspended and resumed at many points (see *coroutine*). await expressions, async for and async with can only be used in the body of a coroutine function.

Functions defined with async def syntax are always coroutine functions, even if they do not contain await or async keywords.

It is a SyntaxError to use a yield from expression inside the body of a coroutine function.

An example of a coroutine function:

```
async def func(param1, param2):
   do_stuff()
   await some_coroutine()
```

Changed in version 3.7: await and async are now keywords; previously they were only treated as such inside the body of a coroutine function.

8.9.2 The async for statement

```
async_for_stmt ::= "async" for_stmt
```

An *asynchronous iterable* provides an __aiter__ method that directly returns an *asynchronous iterator*, which can call asynchronous code in its __anext__ method.

The async for statement allows convenient iteration over asynchronous iterables.

The following code:

```
async for TARGET in ITER:

SUITE
else:

SUITE2
```

Is semantically equivalent to:

```
iter = (ITER)
iter = type(iter).__aiter__(iter)
running = True

while running:
    try:
        TARGET = await type(iter).__anext__(iter)
    except StopAsyncIteration:
        running = False
    else:
        SUITE
else:
    SUITE2
```

See also __aiter__() and __anext__() for details.

It is a SyntaxError to use an async for statement outside the body of a coroutine function.

8.9. Coroutines 127

8.9.3 The async with statement

```
async_with_stmt ::= "async" with_stmt
```

An asynchronous context manager is a context manager that is able to suspend execution in its enter and exit methods.

The following code:

```
async with EXPRESSION as TARGET:
SUITE
```

is semantically equivalent to:

```
manager = (EXPRESSION)
aenter = type (manager) .__aenter__
aexit = type (manager) .__aexit__
value = await aenter (manager)
hit_except = False

try:
    TARGET = value
    SUITE
except:
    hit_except = True
    if not await aexit (manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        await aexit (manager, None, None)
```

See also __aenter__() and __aexit__() for details.

It is a SyntaxError to use an async with statement outside the body of a coroutine function.

```
PEP 492 - Coroutines with async and await syntax

The proposal that made coroutines a proper standalone concept in Python, and added supporting syntax.
```

8.10 Type parameter lists

Added in version 3.12.

Changed in version 3.13: Support for default values was added (see PEP 696).

```
type_params ::= "[" type_param ("," type_param)* "]"
type_param ::= typevar | typevartuple | paramspec
typevar ::= identifier (":" expression)? ("=" expression)?
typevartuple ::= "*" identifier ("=" expression)?
paramspec ::= "**" identifier ("=" expression)?
```

Functions (including coroutines), classes and type aliases may contain a type parameter list:

```
def max[T](args: list[T]) -> T:
    ...
    (continues on next mass)
```

(continues on next page)

(continued from previous page)

```
async def amax[T](args: list[T]) -> T:
    ...

class Bag[T]:
    def __iter__(self) -> Iterator[T]:
    ...

def add(self, arg: T) -> None:
    ...

type ListOrSet[T] = list[T] | set[T]
```

Semantically, this indicates that the function, class, or type alias is generic over a type variable. This information is primarily used by static type checkers, and at runtime, generic objects behave much like their non-generic counterparts.

Type parameters are declared in square brackets ([]) immediately after the name of the function, class, or type alias. The type parameters are accessible within the scope of the generic object, but not elsewhere. Thus, after a declaration $def\ func[T]$ (): pass, the name T is not available in the module scope. Below, the semantics of generic objects are described with more precision. The scope of type parameters is modeled with a special function (technically, an annotation scope) that wraps the creation of the generic object.

Generic functions, classes, and type aliases have a __type_params__ attribute listing their type parameters.

Type parameters come in three kinds:

- typing. TypeVar, introduced by a plain name (e.g., T). Semantically, this represents a single type to a type checker.
- typing. TypeVarTuple, introduced by a name prefixed with a single asterisk (e.g., *Ts). Semantically, this stands for a tuple of any number of types.
- typing.ParamSpec, introduced by a name prefixed with two asterisks (e.g., **P). Semantically, this stands for the parameters of a callable.

typing. TypeVar declarations can define *bounds* and *constraints* with a colon (:) followed by an expression. A single expression after the colon indicates a bound (e.g. T: int). Semantically, this means that the typing. TypeVar can only represent types that are a subtype of this bound. A parenthesized tuple of expressions after the colon indicates a set of constraints (e.g. T: (str, bytes)). Each member of the tuple should be a type (again, this is not enforced at runtime). Constrained type variables can only take on one of the types in the list of constraints.

For typing. TypeVars declared using the type parameter list syntax, the bound and constraints are not evaluated when the generic object is created, but only when the value is explicitly accessed through the attributes __bound_ and __constraints_. To accomplish this, the bounds or constraints are evaluated in a separate *annotation scope*.

 ${\tt typing.TypeVarTuples} \ and \ {\tt typing.ParamSpecs} \ cannot \ have \ bounds \ or \ constraints.$

All three flavors of type parameters can also have a *default value*, which is used when the type parameter is not explicitly provided. This is added by appending a single equals sign (=) followed by an expression. Like the bounds and constraints of type variables, the default value is not evaluated when the object is created, but only when the type parameter's __default__ attribute is accessed. To this end, the default value is evaluated in a separate *annotation scope*. If no default value is specified for a type parameter, the __default__ attribute is set to the special sentinel object typing.NoDefault.

The following example indicates the full set of allowed type parameter declarations:

```
def overly_generic[
    SimpleTypeVar,
    TypeVarWithDefault = int,
    TypeVarWithBound: int,
    TypeVarWithConstraints: (str, bytes),
    *SimpleTypeVarTuple = (int, float),
    (continues on next page)
```

(continued from previous page)

```
**SimpleParamSpec = (str, bytearray),

[](
    a: SimpleTypeVar,
    b: TypeVarWithDefault,
    c: TypeVarWithBound,
    d: Callable[SimpleParamSpec, TypeVarWithConstraints],
    *e: SimpleTypeVarTuple,
): ...
```

8.10.1 Generic functions

Generic functions are declared as follows:

```
def func[T] (arg: T): ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_func():
    T = typing.TypeVar("T")
    def func(arg: T): ...
    func.__type_params__ = (T,)
    return func
func = TYPE_PARAMS_OF_func()
```

Here annotation-def indicates an *annotation scope*, which is not actually bound to any name at runtime. (One other liberty is taken in the translation: the syntax does not go through attribute access on the typing module, but creates an instance of typing. TypeVar directly.)

The annotations of generic functions are evaluated within the annotation scope used for declaring the type parameters, but the function's defaults and decorators are not.

The following example illustrates the scoping rules for these cases, as well as for additional flavors of type parameters:

```
@decorator
def func[T: int, *Ts, **P](*args: *Ts, arg: Callable[P, T] = some_default):
    ...
```

Except for the lazy evaluation of the TypeVar bound, this is equivalent to:

```
DEFAULT_OF_arg = some_default
annotation-def TYPE_PARAMS_OF_func():
    annotation-def BOUND_OF_T():
        return int
    # In reality, BOUND_OF_T() is evaluated only on demand.
    T = typing.TypeVar("T", bound=BOUND_OF_T())

Ts = typing.TypeVarTuple("Ts")
    P = typing.ParamSpec("P")

def func(*args: *Ts, arg: Callable[P, T] = DEFAULT_OF_arg):
        ...
    func.__type_params__ = (T, Ts, P)
    return func
func = decorator(TYPE_PARAMS_OF_func())
```

The capitalized names like <code>DEFAULT_OF_arg</code> are not actually bound at runtime.

8.10.2 Generic classes

Generic classes are declared as follows:

```
class Bag[T]: ...
```

This syntax is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(typing.Generic[T]):
        __type_params__ = (T,)
        ...
    return Bag
Bag = TYPE_PARAMS_OF_Bag()
```

Here again annotation-def (not a real keyword) indicates an *annotation scope*, and the name TYPE_PARAMS_OF_Bag is not actually bound at runtime.

Generic classes implicitly inherit from typing. Generic. The base classes and keyword arguments of generic classes are evaluated within the type scope for the type parameters, and decorators are evaluated outside that scope. This is illustrated by this example:

```
@decorator
class Bag(Base[T], arg=T): ...
```

This is equivalent to:

```
annotation-def TYPE_PARAMS_OF_Bag():
    T = typing.TypeVar("T")
    class Bag(Base[T], typing.Generic[T], arg=T):
        __type_params__ = (T,)
        ...
    return Bag
Bag = decorator(TYPE_PARAMS_OF_Bag())
```

8.10.3 Generic type aliases

The type statement can also be used to create a generic type alias:

```
type ListOrSet[T] = list[T] | set[T]
```

Except for the *lazy evaluation* of the value, this is equivalent to:

```
annotation-def TYPE_PARAMS_OF_ListOrSet():
    T = typing.TypeVar("T")

annotation-def VALUE_OF_ListOrSet():
    return list[T] | set[T]
# In reality, the value is lazily evaluated
    return typing.TypeAliasType("ListOrSet", VALUE_OF_ListOrSet(), type_params=(T,
    ))
ListOrSet = TYPE_PARAMS_OF_ListOrSet()
```

Here, annotation-def (not a real keyword) indicates an *annotation scope*. The capitalized names like TYPE_PARAMS_OF_ListOrSet are not actually bound at runtime.

CHAPTER

NINE

TOP-LEVEL COMPONENTS

The Python interpreter can get its input from a number of sources: from a script passed to it as standard input or as program argument, typed in interactively, from a module source file, etc. This chapter gives the syntax used in these cases.

9.1 Complete Python programs

While a language specification need not prescribe how the language interpreter is invoked, it is useful to have a notion of a complete Python program. A complete Python program is executed in a minimally initialized environment: all built-in and standard modules are available, but none have been initialized, except for sys (various system services), builtins (built-in functions, exceptions and None) and __main__. The latter is used to provide the local and global namespace for execution of the complete program.

The syntax for a complete Python program is that for file input, described in the next section.

The interpreter may also be invoked in interactive mode; in this case, it does not read and execute a complete program but reads and executes one statement (possibly compound) at a time. The initial environment is identical to that of a complete program; each statement is executed in the namespace of __main__.

A complete program can be passed to the interpreter in three forms: with the -c string command line option, as a file passed as the first command line argument, or as standard input. If the file or standard input is a tty device, the interpreter enters interactive mode; otherwise, it executes the file as a complete program.

9.2 File input

All input read from non-interactive files has the same form:

```
file_input ::= (NEWLINE | statement) *
```

This syntax is used in the following situations:

- when parsing a complete Python program (from a file or from a string);
- when parsing a module;
- when parsing a string passed to the exec() function;

9.3 Interactive input

Input in interactive mode is parsed using the following grammar:

```
interactive input ::= [stmt list] NEWLINE | compound stmt NEWLINE
```

Note that a (top-level) compound statement must be followed by a blank line in interactive mode; this is needed to help the parser detect the end of the input.

9.4 Expression input

eval() is used for expression input. It ignores leading whitespace. The string argument to eval() must have the following form:

eval_input ::= expression_list NEWLINE*

FULL GRAMMAR SPECIFICATION

This is the full Python grammar, derived directly from the grammar used to generate the CPython parser (see Grammar/python.gram). The version here omits details related to code generation and error recovery.

The notation is a mixture of EBNF and PEG. In particular, & followed by a symbol, token or parenthesized group indicates a positive lookahead (i.e., is required to match but not consumed), while ! indicates a negative lookahead (i.e., is required *not* to match). We use the | separator to mean PEG's "ordered choice" (written as / in traditional PEG grammars). See **PEP 617** for more details on the grammar's syntax.

```
# PEG grammar for Python
              ======== START OF THE GRAMMAR ==================
# General grammatical elements and rules:
 * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
     - These rules are NOT used in the first pass of the parser.
     - Only if the first pass fails to parse, a second pass including the invalid
       rules will be executed.
      - If the parser fails in the second phase with a generic syntax error, the
       location of the generic failure of the first pass will be used (this avoids
       reporting incorrect locations due to the invalid rules).
      - The order of the alternatives involving invalid rules matter
       (like any rule in PEG).
# Grammar Syntax (see PEP 617 for more information):
# rule_name: expression
   Optionally, a type can be included right after the rule name, which
   specifies the return type of the C or Python function corresponding to the
   rule:
# rule_name[return_type]: expression
  If the return type is omitted, then a void * is returned in C and an Any in
  Python.
# e1 e2
  Match e1, then match e2.
# e1 / e2
  Match e1 or e2.
   The first alternative can also appear on the line after the rule name for
   formatting purposes. In that case, a | must be used before the first
#
   alternative, like so:
```

(continues on next page)

(continued from previous page)

```
rule_name[return_type]:
#
           | first_alt
             | second_alt
# ( e )
  Match e (allows also to use other operators in the group like '(e) *')
# [ e ] or e?
# Optionally match e.
# e*
  Match zero or more occurrences of e.
# e+
  Match one or more occurrences of e.
  Match one or more occurrences of e, separated by s. The generated parse tree
\# does not include the separator. This is otherwise identical to (e (s e)*).
# Succeed if e can be parsed, without consuming any input.
# !e
  Fail if e can be parsed, without consuming any input.
  Commit to the current alternative, even if it fails to parse.
# Eager parse e. The parser will not backtrack and will immediately
  fail with SyntaxError if e cannot be parsed.
# STARTING RULES
# -----
file: [statements] ENDMARKER
interactive: statement_newline
eval: expressions NEWLINE* ENDMARKER
func_type: '(' [type_expressions] ')' '->' expression NEWLINE* ENDMARKER
# GENERAL STATEMENTS
# -----
statements: statement+
statement: compound_stmt | simple_stmts
statement_newline:
   | compound_stmt NEWLINE
   | simple_stmts
   | NEWLINE
   | ENDMARKER
simple_stmts:
   | simple_stmt !';' NEWLINE # Not needed, there for speedup
   | ';'.simple_stmt+ [';'] NEWLINE
# NOTE: assignment MUST precede expression, else parsing a simple assignment
# will throw a SyntaxError.
simple_stmt:
   | assignment
   | type_alias
   | star_expressions
```

(continues on next page)

```
| return_stmt
    | import_stmt
    | raise_stmt
    | 'pass'
   | del_stmt
   | yield_stmt
   | assert_stmt
    | 'break'
   | 'continue'
    | global_stmt
    | nonlocal_stmt
compound_stmt:
  | function_def
   | if_stmt
   | class_def
   | with_stmt
   | for_stmt
   | try_stmt
   | while_stmt
   | match_stmt
# SIMPLE STATEMENTS
# -----
# NOTE: annotated_rhs may start with 'yield'; yield_expr must start with 'yield'
assignment:
  | NAME ':' expression ['=' annotated_rhs ]
   | ('(' single_target ')'
        | single_subscript_attribute_target) ':' expression ['=' annotated_rhs ]
   | (star_targets '=' )+ (yield_expr | star_expressions) !'=' [TYPE_COMMENT]
    | single_target augassign ~ (yield_expr | star_expressions)
annotated_rhs: yield_expr | star_expressions
augassign:
  | '+='
   | '-='
   | '*='
   | '@='
   | '/='
    | '%='
    | '&='
   | '|='
   | '^='
   | '<<='
   | '>>='
   | '**='
    | '//='
return_stmt:
   | 'return' [star_expressions]
raise_stmt:
   | 'raise' expression ['from' expression ]
    | 'raise'
```

```
global_stmt: 'global' ','.NAME+
nonlocal_stmt: 'nonlocal' ','.NAME+
del_stmt:
  | 'del' del_targets &(';' | NEWLINE)
yield_stmt: yield_expr
assert_stmt: 'assert' expression [',' expression ]
import_stmt:
  | import_name
   | import_from
# Import statements
import_name: 'import' dotted_as_names
# note below: the ('.' | '...') is necessary because '...' is tokenized as ELLIPSIS
import_from:
   | 'from' ('.' | '...') * dotted_name 'import' import_from_targets
   | 'from' ('.' | '...') + 'import' import_from_targets
import_from_targets:
   | '(' import_from_as_names [','] ')'
   | import_from_as_names !','
import_from_as_names:
  | ','.import_from_as_name+
import_from_as_name:
  | NAME ['as' NAME ]
dotted_as_names:
  | ','.dotted_as_name+
dotted_as_name:
   | dotted_name ['as' NAME ]
dotted_name:
  | dotted_name '.' NAME
   | NAME
# COMPOUND STATEMENTS
# ==========
# Common elements
   | NEWLINE INDENT statements DEDENT
    | simple_stmts
decorators: ('@' named_expression NEWLINE )+
# Class definitions
class_def:
```

```
| decorators class_def_raw
   | class_def_raw
class_def_raw:
   | 'class' NAME [type_params] ['(' [arguments] ')' ] ':' block
# Function definitions
function_def:
  | decorators function_def_raw
   | function_def_raw
function_def_raw:
  | 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':' [func_type_
→comment] block
  | 'async' 'def' NAME [type_params] '(' [params] ')' ['->' expression ] ':'_
→[func_type_comment] block
# Function parameters
params:
  | parameters
parameters:
   | slash_no_default param_no_default* param_with_default* [star_etc]
   | slash_with_default param_with_default* [star_etc]
   | param_no_default+ param_with_default* [star_etc]
   | param_with_default+ [star_etc]
   | star_etc
# Some duplication here because we can't write (',' | &')'),
# which is because we don't support empty alternatives (yet).
slash no default:
  | param_no_default+ '/' ','
   | param_no_default+ '/' &')'
slash_with_default:
    | param_no_default* param_with_default+ '/' ','
    | param_no_default* param_with_default+ '/' &')'
star_etc:
   | '*' param_no_default param_maybe_default* [kwds]
   | '*' param_no_default_star_annotation param_maybe_default* [kwds]
   | '*' ',' param_maybe_default+ [kwds]
   | kwds
kwds:
   | '**' param_no_default
# One parameter. This *includes* a following comma and type comment.
# There are three styles:
# - No default
# - With default
```

```
# - Maybe with default
# There are two alternative forms of each, to deal with type comments:
# - Ends in a comma followed by an optional type comment
# - No comma, optional type comment, must be followed by close paren
# The latter form is for a final parameter without trailing comma.
param_no_default:
   | param ',' TYPE_COMMENT?
   | param TYPE_COMMENT? &')'
param_no_default_star_annotation:
   param_star_annotation ',' TYPE_COMMENT?
   | param_star_annotation TYPE_COMMENT? &')'
param_with_default:
   | param default ',' TYPE_COMMENT?
   | param default TYPE_COMMENT? &')'
param_maybe_default:
    | param default? ',' TYPE_COMMENT?
   | param default? TYPE_COMMENT? &')'
param: NAME annotation?
param_star_annotation: NAME star_annotation
annotation: ':' expression
star_annotation: ':' star_expression
default: '=' expression | invalid_default
# If statement
if_stmt:
   | 'if' named_expression ':' block elif_stmt
   | 'if' named_expression ':' block [else_block]
elif_stmt:
   | 'elif' named_expression ':' block elif_stmt
    'elif' named_expression ':' block [else_block]
else block:
  | 'else' ':' block
# While statement
while_stmt:
  | 'while' named_expression ':' block [else_block]
# For statement
  | 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block [else_
  | 'async' 'for' star_targets 'in' ~ star_expressions ':' [TYPE_COMMENT] block_
\hookrightarrow [else_block]
# With statement
```

```
with_stmt:
   | 'with' '(' ','.with_item+ ','? ')' ':' [TYPE_COMMENT] block
    | 'with' ','.with_item+ ':' [TYPE_COMMENT] block
    | 'async' 'with' '(' ','.with_item+ ','? ')' ':' block
    | 'async' 'with' ','.with_item+ ':' [TYPE_COMMENT] block
with_item:
   | expression 'as' star_target &(',' | ')' | ':')
   expression
# Try statement
try_stmt:
  | 'try' ':' block finally_block
   | 'try' ':' block except_block+ [else_block] [finally_block]
    | 'try' ':' block except_star_block+ [else_block] [finally_block]
# Except statement
except_block:
  | 'except' expression ['as' NAME ] ':' block
   | 'except' ':' block
except_star_block:
   | 'except' '*' expression ['as' NAME ] ':' block
finally_block:
  | 'finally' ':' block
# Match statement
match_stmt:
   | "match" subject_expr ':' NEWLINE INDENT case_block+ DEDENT
subject_expr:
   | star_named_expression ',' star_named_expressions?
    | named_expression
case_block:
   | "case" patterns guard? ':' block
guard: 'if' named_expression
patterns:
   | open_sequence_pattern
   | pattern
pattern:
   | as_pattern
   | or_pattern
as_pattern:
   | or_pattern 'as' pattern_capture_target
```

```
or_pattern:
   | '|'.closed_pattern+
closed_pattern:
  | literal_pattern
   | capture_pattern
   | wildcard_pattern
   | value_pattern
   | group_pattern
   | sequence_pattern
   | mapping_pattern
   | class_pattern
# Literal patterns are used for equality and identity constraints
literal_pattern:
   | signed_number !('+' | '-')
   | complex_number
   | strings
    | 'None'
   | 'True'
   | 'False'
# Literal expressions are used to restrict permitted mapping pattern keys
literal_expr:
   | signed_number !('+' | '-')
   | complex_number
   | strings
   | 'None'
   | 'True'
   | 'False'
complex_number:
   | signed_real_number '+' imaginary_number
    | signed_real_number '-' imaginary_number
signed_number:
  | NUMBER
   | '-' NUMBER
signed_real_number:
   | real_number
   | '-' real_number
real_number:
  | NUMBER
imaginary_number:
  | NUMBER
capture_pattern:
  | pattern_capture_target
pattern_capture_target:
  | !"_" NAME !('.' | '(' | '=')
wildcard_pattern:
```

```
| "_"
value_pattern:
  | attr !('.' | '(' | '=')
attr:
  | name_or_attr '.' NAME
name_or_attr:
   | attr
   | NAME
group_pattern:
  | '(' pattern ')'
sequence_pattern:
   | '[' maybe_sequence_pattern? ']'
    '(' open_sequence_pattern? ')'
open_sequence_pattern:
   | maybe_star_pattern ',' maybe_sequence_pattern?
maybe_sequence_pattern:
  | ','.maybe_star_pattern+ ','?
maybe_star_pattern:
   | star_pattern
   | pattern
star_pattern:
   | '*' pattern_capture_target
    | '*' wildcard_pattern
mapping_pattern:
   | '{' '}'
    | '{' double_star_pattern ','? '}'
   | '{' items_pattern ',' double_star_pattern ','? '}'
   | '{' items_pattern ','? '}'
items_pattern:
   | ','.key_value_pattern+
key_value_pattern:
   | (literal_expr | attr) ':' pattern
double_star_pattern:
  | '**' pattern_capture_target
class_pattern:
   | name_or_attr '(' ')'
    | name_or_attr '(' positional_patterns ','? ')'
   | name_or_attr '(' keyword_patterns ','? ')'
   | name_or_attr '(' positional_patterns ',' keyword_patterns ','? ')'
positional_patterns:
  | ','.pattern+
```

```
keyword_patterns:
   | ','.keyword_pattern+
keyword_pattern:
  | NAME '=' pattern
# Type statement
type_alias:
   | "type" NAME [type_params] '=' expression
# Type parameter declaration
type_params:
   | invalid_type_params
    | '[' type_param_seq ']'
type_param_seq: ','.type_param+ [',']
type_param:
   | NAME [type_param_bound] [type_param_default]
   | '*' NAME [type_param_starred_default]
    | '**' NAME [type_param_default]
type_param_bound: ':' expression
type_param_default: '=' expression
type_param_starred_default: '=' star_expression
# EXPRESSIONS
expressions:
   | expression (',' expression )+ [',']
   | expression ','
   | expression
expression:
   | disjunction 'if' disjunction 'else' expression
    | disjunction
    | lambdef
yield_expr:
  | 'yield' 'from' expression
    | 'yield' [star_expressions]
star_expressions:
   | star_expression (',' star_expression )+ [',']
    | star_expression ','
   | star_expression
star_expression:
  | '*' bitwise_or
   | expression
```

```
star_named_expressions: ','.star_named_expression+ [',']
star_named_expression:
   | '*' bitwise_or
   | named_expression
assignment_expression:
  | NAME ':=' ~ expression
named_expression:
   | assignment_expression
    | expression !':='
disjunction:
   | conjunction ('or' conjunction )+
    | conjunction
conjunction:
    | inversion ('and' inversion )+
   | inversion
inversion:
  | 'not' inversion
   | comparison
# Comparison operators
comparison:
   | bitwise_or compare_op_bitwise_or_pair+
    | bitwise_or
compare_op_bitwise_or_pair:
   | eq_bitwise_or
    | noteq_bitwise_or
   | lte_bitwise_or
   | lt_bitwise_or
   | gte_bitwise_or
   | gt_bitwise_or
   | notin_bitwise_or
    | in_bitwise_or
   | isnot_bitwise_or
    | is_bitwise_or
eq_bitwise_or: '==' bitwise_or
noteq_bitwise_or:
   | ('!=' ) bitwise_or
lte_bitwise_or: '<=' bitwise_or</pre>
lt_bitwise_or: '<' bitwise_or</pre>
gte_bitwise_or: '>=' bitwise_or
gt_bitwise_or: '>' bitwise_or
notin_bitwise_or: 'not' 'in' bitwise_or
in_bitwise_or: 'in' bitwise_or
isnot_bitwise_or: 'is' 'not' bitwise_or
is_bitwise_or: 'is' bitwise_or
```

```
# Bitwise operators
bitwise_or:
   | bitwise_or '|' bitwise_xor
   | bitwise_xor
bitwise_xor:
   | bitwise_xor '^' bitwise_and
    | bitwise_and
bitwise_and:
  | bitwise_and '&' shift_expr
   | shift_expr
shift_expr:
   | shift_expr '<<' sum
   | shift_expr '>>' sum
   sum
# Arithmetic operators
sum:
  | sum '+' term
   | sum '-' term
   | term
term:
  | term '*' factor
   | term '/' factor
   | term '//' factor
   | term '%' factor
   | term '@' factor
   | factor
factor:
  | '+' factor
   | '-' factor
   | '~' factor
   | power
power:
  | await_primary '**' factor
   | await_primary
# Primary elements
# Primary elements are things like "obj.something.something", "obj[something]",
→"obj(something)", "obj" ...
await_primary:
  | 'await' primary
   | primary
```

```
primary:
   | primary '.' NAME
    | primary genexp
    | primary '(' [arguments] ')'
    | primary '[' slices ']'
    | atom
slices:
    | slice !','
    | ','.(slice | starred_expression)+ [',']
slice:
   | [expression] ':' [expression] [':' [expression] ]
   | named_expression
atom:
   | NAME
   'True'
   | 'False'
   | 'None'
   | strings
   | NUMBER
   | (tuple | group | genexp)
   | (list | listcomp)
   | (dict | set | dictcomp | setcomp)
   | '...'
group:
  | '(' (yield_expr | named_expression) ')'
# Lambda functions
lambdef:
  | 'lambda' [lambda_params] ':' expression
lambda_params:
  | lambda_parameters
# lambda_parameters etc. duplicates parameters but without annotations
# or type comments, and if there's no comma after a parameter, we expect
# a colon, not a close parenthesis. (For more, see parameters above.)
lambda_parameters:
  | lambda_slash_no_default lambda_param_no_default* lambda_param_with_default*_
→ [lambda_star_etc]
   | lambda_slash_with_default lambda_param_with_default* [lambda_star_etc]
    | lambda_param_no_default+ lambda_param_with_default* [lambda_star_etc]
    | lambda_param_with_default+ [lambda_star_etc]
    | lambda_star_etc
lambda_slash_no_default:
   | lambda_param_no_default+ '/' ','
    | lambda_param_no_default+ '/' &':'
```

```
lambda_slash_with_default:
    | lambda_param_no_default* lambda_param_with_default+ '/' ','
    | lambda_param_no_default* lambda_param_with_default+ '/' &':'
lambda_star_etc:
    | '*' lambda_param_no_default lambda_param_maybe_default* [lambda_kwds]
    | '*' ',' lambda_param_maybe_default+ [lambda_kwds]
    | lambda_kwds
lambda_kwds:
  | '**' lambda_param_no_default
lambda_param_no_default:
  | lambda_param ','
   | lambda_param &':'
lambda_param_with_default:
   | lambda_param default ','
    | lambda_param default &':'
lambda_param_maybe_default:
   | lambda_param default? ','
   | lambda_param default? &':'
lambda_param: NAME
# LITERALS
# -----
fstring_middle:
  | fstring_replacement_field
   | FSTRING_MIDDLE
fstring_replacement_field:
  | '{' annotated_rhs '='? [fstring_conversion] [fstring_full_format_spec] '}'
fstring_conversion:
  | "!" NAME
fstring_full_format_spec:
   | ':' fstring_format_spec*
fstring_format_spec:
   | FSTRING_MIDDLE
   | fstring_replacement_field
fstring:
   | FSTRING_START fstring_middle* FSTRING_END
string: STRING
strings: (fstring|string)+
list:
   | '[' [star_named_expressions] ']'
tuple:
   '(' [star_named_expression ',' [star_named_expressions] ] ')'
set: '{' star_named_expressions '}'
# Dicts
# ----
dict:
```

```
| '{' [double_starred_kvpairs] '}'
double_starred_kvpairs: ','.double_starred_kvpair+ [',']
double_starred_kvpair:
  | '**' bitwise_or
   | kvpair
kvpair: expression ':' expression
# Comprehensions & Generators
for_if_clauses:
  | for_if_clause+
for if clause:
   | 'async' 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
    | 'for' star_targets 'in' ~ disjunction ('if' disjunction )*
listcomp:
  | '[' named_expression for_if_clauses ']'
setcomp:
   | '{' named_expression for_if_clauses '}'
  | '(' ( assignment_expression | expression !':=') for_if_clauses ')'
dictcomp:
  | '{' kvpair for_if_clauses '}'
# FUNCTION CALL ARGUMENTS
# -----
arguments:
  | args [','] &')'
args:
   | ','.(starred_expression | ( assignment_expression | expression !':=') !'=')+_
→[',' kwargs ]
   | kwargs
kwarqs:
  | ','.kwarg_or_starred+ ',' ','.kwarg_or_double_starred+
   | ','.kwarg_or_starred+
   ','.kwarg_or_double_starred+
starred_expression:
   | '*' expression
kwarg_or_starred:
   | NAME '=' expression
   | starred_expression
kwarg_or_double_starred:
```

```
| NAME '=' expression
    | '**' expression
# ASSIGNMENT TARGETS
# -----
# Generic targets
# NOTE: star_targets may contain *bitwise_or, targets may not.
star_targets:
   | star_target !','
   | star_target (',' star_target )* [',']
star_targets_list_seq: ','.star_target+ [',']
star_targets_tuple_seq:
   | star_target (',' star_target )+ [',']
    | star_target ','
star_target:
  | '*' (!'*' star_target)
   | target_with_star_atom
target_with_star_atom:
   | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom
star_atom:
   | NAME
   | '(' target_with_star_atom ')'
   | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'
single_target:
  | single_subscript_attribute_target
    | NAME
   | '(' single_target ')'
single_subscript_attribute_target:
   | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
t_primary:
   | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
   | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead
    | atom &t_lookahead
t_lookahead: '(' | '[' | '.'
# Targets for del statements
```

```
del_targets: ','.del_target+ [',']
del_target:
   | t_primary '.' NAME !t_lookahead
   | t_primary '[' slices ']' !t_lookahead
   | del_t_atom
del_t_atom:
  | NAME
  | '(' del_target ')'
   | '(' [del_targets] ')'
  | '[' [del_targets] ']'
# TYPING ELEMENTS
# type_expressions allow */** but ignore them
type_expressions:
   | ','.expression+ ',' '*' expression ',' '**' expression
   | ','.expression+ ',' '*' expression
   | ','.expression+ ',' '**' expression
   | '*' expression ',' '**' expression
  | '*' expression
  | '**' expression
   | ','.expression+
func_type_comment:
  | NEWLINE TYPE_COMMENT & (NEWLINE INDENT)  # Must be followed by indented block
  TYPE_COMMENT
```

Α

GLOSSARY

>>>

The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

. . .

Can refer to:

- The default Python prompt of the *interactive* shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
- The Ellipsis built-in constant.

abstract base class

Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like hasattr() would be clumsy or subtly wrong (for example with *magic methods*). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by <code>isinstance()</code> and <code>issubclass()</code>; see the <code>abc</code> module documentation. Python comes with many built-in ABCs for data structures (in the <code>collections.abc</code> module), numbers (in the numbers module), streams (in the <code>io</code> module), import finders and loaders (in the <code>importlib.abc</code> module). You can create your own ABCs with the <code>abc</code> module.

annotation

A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a *type hint*.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the __annotations__ special attribute of modules, classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

argument

A value passed to a function (or method) when calling the function. There are two kinds of argument:

• *keyword argument*: an argument preceded by an identifier (e.g. name=) in a function call or passed as a value in a dictionary preceded by **. For example, 3 and 5 are both keyword arguments in the following calls to complex():

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

• *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an *iterable* preceded by *. For example, 3 and 5 are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the *Calls* section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the *parameter* glossary entry, the FAQ question on the difference between arguments and parameters, and PEP 362.

asynchronous context manager

An object which controls the environment seen in an async with statement by defining __aenter__() and __aexit__() methods. Introduced by PEP 492.

asynchronous generator

A function which returns an *asynchronous generator iterator*. It looks like a coroutine function defined with *async def* except that it contains *yield* expressions for producing a series of values usable in an *async for* loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain await expressions as well as async for, and async with statements.

asynchronous generator iterator

An object created by a asynchronous generator function.

This is an *asynchronous iterator* which when called using the __anext__ () method returns an awaitable object which will execute the body of the asynchronous generator function until the next yield expression.

Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by __anext__(), it picks up where it left off. See PEP 492 and PEP 525.

asynchronous iterable

An object, that can be used in an async for statement. Must return an asynchronous iterator from its __aiter__() method. Introduced by PEP 492.

asynchronous iterator

An object that implements the __aiter__() and __anext__() methods. __anext__() must return an awaitable object. async for resolves the awaitables returned by an asynchronous iterator's __anext__() method until it raises a StopAsyncIteration exception. Introduced by PEP 492.

attribute

A value associated with an object which is usually referenced by name using dotted expressions. For example, if an object o has an attribute o it would be referenced as o.o.

It is possible to give an object an attribute whose name is not an identifier as defined by *Identifiers and keywords*, for example using <code>setattr()</code>, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with <code>getattr()</code>.

awaitable

An object that can be used in an await expression. Can be a *coroutine* or an object with an __await__() method. See also **PEP 492**.

BDFL

Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

binary file

A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode ('rb', 'wb' or 'rb+'), sys.stdin.buffer, sys.stdout.buffer, and instances of io.BytesIO and gzip.GzipFile.

See also *text file* for a file object able to read and write str objects.

borrowed reference

In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling Py_INCREF () on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The Py_NewRef () function can be used to create a new *strong reference*.

bytes-like object

An object that supports the bufferobjects and can export a C-contiguous buffer. This includes all bytes, bytearray, and array objects, as well as many common memoryview objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as "read-write bytes-like objects". Example mutable buffer objects include bytearray and a memoryview of a bytearray. Other operations require the binary data to be stored in immutable objects ("read-only bytes-like objects"); examples of these include bytes and a memoryview of a bytes object.

bytecode

Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in .pyc files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This "intermediate language" is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the dis module.

callable

A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the __call_() method is also a callable.

callback

A subroutine function which is passed as an argument to be executed at some point in the future.

class

A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

class variable

A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

closure variable

A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the *nonlocal* keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the inner function in the following code, both x and print are *free variables*, but only x is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the <code>codeobject.co_freevars</code> attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general <code>free variable</code> term is sometimes used even when the intended meaning is to refer specifically to closure variables.

complex number

An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of -1), often written i in mathematics or j in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a j suffix, e.g., 3+1j. To get access to complex equivalents of the math module, use cmath. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

context

This term has different meanings depending on where and how it is used. Some common meanings:

- The temporary state or environment established by a *context manager* via a with statement.
- The collection of keyvalue bindings associated with a particular contextvars. Context object and accessed via ContextVar objects. Also see *context variable*.
- A contextvars.Context object. Also see current context.

context management protocol

The __enter__() and __exit__() methods called by the with statement. See PEP 343.

context manager

An object which implements the *context management protocol* and controls the environment seen in a with statement. See PEP 343.

context variable

A variable whose value depends on which context is the *current context*. Values are accessed via contextvars.ContextVar objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguous

A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

coroutine

Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the <code>async def</code> statement. See also **PEP 492**.

coroutine function

A function which returns a *coroutine* object. A coroutine function may be defined with the *async def* statement, and may contain *await*, *async for*, and *async with* keywords. These were introduced by **PEP** 492.

CPython

The canonical implementation of the Python programming language, as distributed on python.org. The term "CPython" is used when necessary to distinguish this implementation from others such as Jython or IronPython.

current context

The *context* (contextvars.Context object) that is currently used by ContextVar objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see asyncio) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

decorator

A function returning another function, usually applied as a function transformation using the @wrapper syntax. Common examples for decorators are classmethod() and staticmethod().

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for *function definitions* and *class definitions* for more about decorators.

descriptor

Any object which defines the methods $_get_()$, $_set_()$, or $_delete_()$. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using a.b to get, set or delete an attribute looks up the object named b in the class dictionary for a, but if b is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see *Implementing Descriptors* or the Descriptor How To Guide.

dictionary

An associative array, where arbitrary keys are mapped to values. The keys can be any object with __hash__ () and __eq__ () methods. Called a hash in Perl.

dictionary comprehension

A compact way to process all or part of the elements in an iterable and return a dictionary with the results. results = {n: n ** 2 for n in range(10)} generates a dictionary containing key n mapped to value n ** 2. See Displays for lists, sets and dictionaries.

dictionary view

The objects returned from dict.keys(), dict.values(), and dict.items() are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use list(dictview). See 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

A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using type() or isinstance(). (Note, however, that duck-typing can be complemented with abstract base classes.) Instead, it typically employs hasattr() tests or EAFP programming.

EAFP

Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements. The technique contrasts with the LBYL style common to many other languages such as C.

expression

A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statements* which cannot be used as expressions, such as *while*. Assignments are also statements, not expressions.

extension module

A module written in C or C++, using Python's C API to interact with the core and with user code.

f-string

String literals prefixed with 'f' or 'F' are commonly called "f-strings" which is short for *formatted string* literals. See also **PEP 498**.

file object

An object exposing a file-oriented API (with methods such as read() or write()) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw *binary files*, buffered *binary files* and *text files*. Their interfaces are defined in the io module. The canonical way to create a file object is by using the open () function.

file-like object

A synonym for file object.

filesystem encoding and error handler

Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise UnicodeError.

The sys.getfilesystemencoding() and sys.getfilesystemencodeerrors() functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the PyConfig_Read() function: see filesystem_encoding and filesystem_errors members of PyConfig.

See also the *locale encoding*.

finder

An object that tries to find the *loader* for a module that is being imported.

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 Finders and loaders and importlib for much more detail.

floor division

Mathematical division that rounds down to nearest integer. The floor division operator is //. For example, the expression 11 // 4 evaluates to 2 in contrast to the 2.75 returned by float true division. Note that (-11) // 4 is -3 because that is -2.75 rounded *downward*. See **PEP 238**.

free threading

A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable

Formally, as defined in the *language execution model*, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the *codeobject.co_freevars* attribute, the term is also sometimes used as a synonym for *closure variable*.

function

A series of statements which returns some value to a caller. It can also be passed zero or more *arguments* which may be used in the execution of the body. See also *parameter*, *method*, and the *Function definitions* section.

function annotation

An annotation of a function parameter or return value.

Function annotations are usually used for *type hints*: for example, this function is expected to take two int arguments and is also expected to have an int return value:

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

Function annotation syntax is explained in section Function definitions.

See *variable annotation* and **PEP 484**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

future

A *future statement*, from __future__ import <feature>, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The __future_ module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection

The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the gc module.

generator

A function which returns a *generator iterator*. It looks like a normal function except that it contains yield expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the next() function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

generator iterator

An object created by a generator function.

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 expression

An *expression* that returns an *iterator*. It looks like a normal expression followed by a for clause defining a loop variable, range, and an optional if clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))  # sum of squares 0, 1, 4, ... 81
285
```

generic function

A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the single dispatch glossary entry, the functools.singledispatch() decorator, and PEP 443.

generic type

A type that can be parameterized; typically a container class such as list or dict. Used for type hints and annotations.

For more details, see generic alias types, PEP 483, PEP 484, PEP 585, and the typing module.

GIL

See global interpreter lock.

global interpreter lock

The mechanism used by the *CPython* interpreter to assure that only one thread executes Python *bytecode* at a time. This simplifies the CPython implementation by making the object model (including critical built-in

types such as dict) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

As of Python 3.13, the GIL can be disabled using the <code>--disable-gil</code> build configuration. After building Python with this option, code must be run with <code>-X gil=0</code> or after setting the <code>PYTHON_GIL=0</code> environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

hash-based pvc

A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See *Cached bytecode invalidation*.

hashable

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a __hash__() method), and can be compared to other objects (it needs an __eq__() method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

Most of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their id().

IDLE

An Integrated Development and Learning Environment for Python. idle is a basic editor and interpreter environment which ships with the standard distribution of Python.

immortal

Immortal objects are a CPython implementation detail introduced in PEP 683.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, True and None are immortal in CPython.

immutable

An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

import path

A list of locations (or *path entries*) that are searched by the *path based finder* for modules to import. During import, this list of locations usually comes from sys.path, but for subpackages it may also come from the parent package's __path__ attribute.

importing

The process by which Python code in one module is made available to Python code in another module.

importer

An object that both finds and loads a module; both a *finder* and *loader* object.

interactive

Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch python with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember help(x)). For more on interactive mode, see tut-interac.

interpreted

Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly

creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

interpreter shutdown

When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the *garbage collector*. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the __main__ module or the script being run has finished executing.

iterable

An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as list, str, and tuple) and some non-sequence types like dict, *file objects*, and objects of any classes you define with an __iter__() method or with a __getitem__() method that implements *sequence* semantics.

Iterables can be used in a for loop and in many other places where a sequence is needed (zip(), map(), ...). When an iterable object is passed as an argument to the built-in function iter(), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() or deal with iterator objects yourself. The for statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, sequence, and generator.

iterator

An object representing a stream of data. Repeated calls to the iterator's __next__() method (or passing it to the built-in function next()) return successive items in the stream. When no more data are available a StopIteration exception is raised instead. At this point, the iterator object is exhausted and any further calls to its __next__() method just raise StopIteration again. Iterators are required to have an __iter__() method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a list) produces a fresh new iterator each time you pass it to the iter() function or use it in a for loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in typeiter.

CPython implementation detail: CPython does not consistently apply the requirement that an iterator define __iter__(). And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

key function

A key function or collation function is a callable that returns a value used for sorting or ordering. For example, locale.strxfrm() is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include min(), max(), sorted(), list.sort(), heapq.merge(), heapq.nsmallest(), heapq.nlargest(), and itertools.groupby().

There are several ways to create a key function. For example, the str.lower() method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a lambda expression such as lambda r: (r[0], r[2]). Also, operator.attrgetter(), operator.itemgetter(), and operator.methodcaller() are three key function constructors. See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument

See argument.

lambda

An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is lambda [parameters]: expression

LBYL

Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many if statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code, if key in mapping: return mapping[key] can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

list

A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is O(1).

list comprehension

A compact way to process all or part of the elements in a sequence and return a list with the results. result = [' $\{:\#04x\}$ '.format(x) for x in range(256) if x % 2 == 0] generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The *if* clause is optional. If omitted, all elements in range(256) are processed.

loader

An object that loads a module. It must define a method named load_module(). A loader is typically returned by a *finder*. See also:

- Finders and loaders
- importlib.abc.Loader
- PEP 302

locale encoding

On Unix, it is the encoding of the LC_CTYPE locale. It can be set with locale.setlocale(locale. LC_CTYPE , new_locale).

On Windows, it is the ANSI code page (ex: "cp1252").

On Android and VxWorks, Python uses "utf-8" as the locale encoding.

locale.getencoding() can be used to get the locale encoding.

See also the filesystem encoding and error handler.

magic method

An informal synonym for special method.

mapping

A container object that supports arbitrary key lookups and implements the methods specified in the collections.abc.Mapping or collections.abc.MutableMapping abstract base classes. Examples include dict, collections.defaultdict, collections.OrderedDict and collections. Counter.

meta path finder

A *finder* returned by a search of sys.meta_path. Meta path finders are related to, but different from *path* entry finders.

See importlib.abc.MetaPathFinder for the methods that meta path finders implement.

metaclass

The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in Metaclasses.

method

A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first *argument* (which is usually called self). See *function* and *nested scope*.

method resolution order

Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

module

An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of *importing*.

See also package.

module spec

A namespace containing the import-related information used to load a module. An instance of importlib. machinery.ModuleSpec.

See also Module specs.

MRO

See method resolution order.

mutable

Mutable objects can change their value but keep their id(). See also immutable.

named tuple

The term "named tuple" applies to any type or class that inherits from tuple and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by time.localtime() and os. stat(). Another example is sys.float_info:

```
>>> sys.float_info[1]  # indexed access
1024
>>> sys.float_info.max_exp  # named field access
1024
>>> isinstance(sys.float_info, tuple)  # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from tuple and that defines named fields. Such a class can be written by hand, or it can be created by inheriting typing. NamedTuple, or with the factory function collections.namedtuple(). The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

namespace

The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions builtins.open and os.open() are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing random.seed() or itertools.islice() makes it clear that those functions are implemented by the random and itertools modules, respectively.

namespace package

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.

See also module.

nested scope

The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference

and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The nonlocal allows writing to outer scopes.

new-style class

Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like __slots__, descriptors, properties, __getattribute__(), class methods, and static methods.

object

Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any *new-style class*.

optimized scope

A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package

A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with a __path__ attribute.

See also regular package and namespace package.

parameter

A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

• positional-or-keyword: specifies an argument that can be passed either positionally or as a keyword argument. This is the default kind of parameter, for example foo and bar in the following:

```
def func(foo, bar=None): ...
```

• *positional-only*: specifies an argument that can be supplied only by position. Positional-only parameters can be defined by including a / character in the parameter list of the function definition after them, for example *posonly1* and *posonly2* in the following:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

• *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare * in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

• *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with *, for example *args* in the following:

```
def func(*args, **kwargs): ...
```

• *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with **, for example *kwargs* in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, the inspect.Parameter class, the *Function definitions* section, and PEP 362.

path entry

A single location on the *import path* which the *path based finder* consults to find modules for importing.

path entry finder

A *finder* returned by a callable on sys.path_hooks (i.e. a *path entry hook*) which knows how to locate modules given a *path entry*.

See importlib.abc.PathEntryFinder for the methods that path entry finders implement.

path entry hook

A callable on the sys.path_hooks list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

path based finder

One of the default meta path finders which searches an import path for modules.

path-like object

An object representing a file system path. A path-like object is either a str or bytes object representing a path, or an object implementing the os.PathLike protocol. An object that supports the os.PathLike protocol can be converted to a str or bytes file system path by calling the os.fspath() function; os.fsdecode() and os.fsencode() can be used to guarantee a str or bytes result instead, respectively. Introduced by PEP 519.

PEP

Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See PEP 1.

portion

A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in PEP 420.

positional argument

See argument.

provisional API

A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a "solution of last resort" - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See PEP 411 for more details.

provisional package

See provisional API.

Python 3000

Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated "Py3k".

Pythonic

An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a for statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print(food[i])
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
   print(piece)
```

qualified name

A dotted name showing the "path" from a module's global scope to a class, function or method defined in that module, as defined in **PEP 3155**. For top-level functions and classes, the qualified name is the same as the object's name:

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. email.mime.text:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count

The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never 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.

regular package

A traditional *package*, such as a directory containing an __init__.py file.

See also namespace package.

REPL

An acronym for the "read-eval-print loop", another name for the interactive interpreter shell.

__slots_

A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence

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 *hashable* keys rather than integers.

The collections.abc.Sequence abstract base class defines a much richer interface that goes beyond just __getitem__() and __len__(), adding count(), index(), __contains__(), and __reversed__().

Types that implement this expanded interface can be registered explicitly using register(). For more documentation on sequence methods generally, see Common Sequence Operations.

set comprehension

A compact way to process all or part of the elements in an iterable and return a set with the results. results = {c for c in 'abracadabra' if c not in 'abc'} generates the set of strings {'r', 'd'}. See Displays for lists, sets and dictionaries.

single dispatch

A form of generic function dispatch where the implementation is chosen based on the type of a single argument.

slice

An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, [] with colons between numbers when several are given, such as in variable_name[1:3:5]. The bracket (subscript) notation uses slice objects internally.

soft deprecated

A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387: Soft Deprecation.

special method

A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in *Special method names*.

statement

A statement is part of a suite (a "block" of code). A statement is either an *expression* or one of several constructs with a keyword, such as *if*, *while* or *for*.

static type checker

An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the typing module.

strong reference

In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling Py_INCREF() when the reference is created and released with Py_DECREF() when the reference is deleted.

The $Py_NewRef()$ function can be used to create a strong reference to an object. Usually, the $Py_DECREF()$ function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

See also borrowed reference.

text encoding

A string in Python is a sequence of Unicode code points (in range U+0000-U+10FFFF). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as "encoding", and recreating the string from the sequence of bytes is known as "decoding".

There are a variety of different text serialization codecs, which are collectively referred to as "text encodings".

text file

A *file object* able to read and write str objects. Often, a text file actually accesses a byte-oriented datastream and handles the *text encoding* automatically. Examples of text files are files opened in text mode ('r' or 'w'), sys.stdin, sys.stdout, and instances of io.StringIO.

See also binary file for a file object able to read and write bytes-like objects.

triple-quoted string

A string which is bound by three instances of either a quotation mark (") or an apostrophe ('). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons.

They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type

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

A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying *type hints*. For example:

could be made more readable like this:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

See typing and PEP 484, which describe this functionality.

type hint

An *annotation* that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using typing.get_type_hints().

See typing and PEP 484, which describe this functionality.

universal newlines

A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention '\n', the Windows convention '\r\n', and the old Macintosh convention '\r'. See **PEP 278** and **PEP 3116**, as well as bytes.splitlines() for an additional use.

variable annotation

An annotation of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
class C:
field: 'annotation'
```

Variable annotations are usually used for *type hints*: for example this variable is expected to take int values:

```
count: int = 0
```

Variable annotation syntax is explained in section Annotated assignment statements.

See *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

virtual environment

A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also venv.

virtual machine

A computer defined entirely in software. Python's virtual machine executes the *bytecode* emitted by the bytecode compiler.

Zen of Python

Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing "import this" at the interactive prompt.

В

ABOUT THESE DOCUMENTS

These documents are generated from reStructuredText sources by Sphinx, a document processor specifically written for the Python documentation.

Development of the documentation and its toolchain is an entirely volunteer effort, just like Python itself. If you want to contribute, please take a look at the reporting-bugs page for information on how to do so. New volunteers are always welcome!

Many thanks go to:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- the Docutils project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 Contributors to the Python Documentation

Many people have contributed to the Python language, the Python standard library, and the Python documentation. See Misc/ACKS in the Python source distribution for a partial list of contributors.

It is only with the input and contributions of the Python community that Python has such wonderful documentation – Thank You!

C

HISTORY AND LICENSE

C.1 History of the software

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 Python-Labs 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.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 and above	2.1.1	2001-now	PSF	yes



GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

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*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.13.0

- 1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.13.0 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 3.13.0 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights Reserved" are retained in Python 3.13.0 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 3.13.0 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 3.13.0.
- 4. PSF is making Python 3.13.0 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 3.13.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
- 5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.13.0 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.13.0, 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 3.13.0, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

- 1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
- 2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
- 3. BeOpen is making the Software available to Licensee on an "AS IS" basis.

 BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF

 EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR

 WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE

 USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
- 4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
- 5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
- 6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.
- 7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

- 1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
- 2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All

Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the internet using the following URL: http://hdl.handle.net/1895.22/1013."

- 3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
- 4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
- 5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
- 6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
- 7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
- 8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright

notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCU-MENTATION

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The $_$ random C extension underlying 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,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

C.3.2 Sockets

The socket module uses the functions, getaddrinfo(), and getnameinfo(), which are coded in separate source files from the WIDE Project, 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.

C.3.3 Asynchronous socket services

The test.support.asynchat and test.support.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.

C.3.4 Cookie management

The http.cookies module contains the following notice:

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.

C.3.5 Execution tracing

The trace module contains the following notice:

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.

C.3.6 UUencode and UUdecode functions

The uu codec contains the following notice:

Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Lance Ellinghouse not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:

- Use binascii module to do the actual line-by-line conversion between ascii and binary. This results in a 1000-fold speedup. The C version is still 5 times faster, though.
- Arguments more compliant with Python standard

C.3.7 XML Remote Procedure Calls

The xmlrpc.client module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-ABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.8 test_epoll

The test.test_epoll module contains the following notice:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.9 Select kqueue

The select module contains the following notice for the kqueue interface:

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.10 SipHash24

The file Python/pyhash.c contains Marek Majkowski' implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013 Marek Majkowski <marek@popcount.org>
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:
The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>
Original location:
  https://github.com/majek/csiphash/
Solution inspired by code from:
  Samuel Neves (supercop/crypto_auth/siphash24/little)
  djb (supercop/crypto_auth/siphash24/little2)
  Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod and dtoa

The file Python/dtoa.c, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

C.3.12 OpenSSL

The modules hashlib, posix and ssl use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
Apache License
                        Version 2.0, January 2004
                     https://www.apache.org/licenses/
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION
1. Definitions.
  "License" shall mean the terms and conditions for use, reproduction,
  and distribution as defined by Sections 1 through 9 of this document.
  "Licensor" shall mean the copyright owner or entity authorized by
  the copyright owner that is granting the License.
  "Legal Entity" shall mean the union of the acting entity and all
  other entities that control, are controlled by, or are under common
  control with that entity. For the purposes of this definition,
  "control" means (i) the power, direct or indirect, to cause the
  direction or management of such entity, whether by contract or
  otherwise, or (ii) ownership of fifty percent (50%) or more of the
  outstanding shares, or (iii) beneficial ownership of such entity.
  "You" (or "Your") shall mean an individual or Legal Entity
  exercising permissions granted by this License.
```

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

- 2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
- 3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You

institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

- 4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
 - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
 - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
 - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
 - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions.
Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

- 6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
- 7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
- 8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
- 9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

C.3.13 expat

The pyexpat extension is built using an included copy of the expat sources unless the build is configured -with-system-expat:

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.14 libffi

The _ctypes C extension underlying the ctypes module 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.

C.3.15 zlib

The zlib extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be

appreciated but is not required.

- 2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
- 3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly Mark Adler

jloup@gzip.org madler@alumni.caltech.edu

C.3.16 cfuhash

The implementation of the hash table used by the tracemalloc is based on the cfuhash project:

Copyright (c) 2005 Don Owens All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.17 libmpdec

The $_decimal\ C$ extension underlying 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.

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the test package (Lib/test/xmltestdata/c14n-20/) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang), All Rights Reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.19 mimalloc

MIT License:

Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.20 asyncio

Parts of the asyncio module are incorporated from uvloop 0.16, which is distributed under the MIT license:

Copyright (c) 2015-2021 MagicStack Inc. http://magic.io

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.21 Global Unbounded Sequences (GUS)

The file Python/qsbr.c is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr smr.c. The file is distributed under the 2-Clause BSD License:

Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions

are met:

- 1. Redistributions of source code must retain the above copyright notice unmodified, this list of conditions, and the following disclaimer.
- 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

APPENDIX

D

COPYRIGHT

Python and this documentation is:

Copyright © 2001-2024 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright @ 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

See *History and License* for complete license and permissions information.

INDEX

Non-alphabetical	in expression lists, 95
, 153	in function calls, 87
ellipsis literal, 18	operator, 89
111	**
string literal, 10	function definition, 124
{ } (curly brackets)	in dictionary displays, 80
dictionary expression, 80	in function calls, 87
in formatted string literal, 12	operator, 88
set expression, 80	**=
. (dot)	augmented assignment, 99
attribute reference, 85	*=
in numeric literal, 15	augmented assignment, 99
! (exclamation)	+ (plus)
in formatted string literal, 12	binary operator, 90
- (minus)	unary operator, 89
binary operator, 90	+=
unary operator, 89	augmented assignment, 99
' (single quote)	, (comma), 78
string literal, 9	argument list, 86
! patterns, 117	expression list, 79, 80, 95, 101, 125
" (double quote)	identifier list, 107
string literal, 9	import statement, 104
"""	in dictionary displays, 80
string literal, 10	in target list, 98
# (hash)	parameter list, 123
comment, 5	slicing, 86
source encoding declaration, 5	with statement, 114
% (percent)	/ (slash)
operator, 89	function definition, 124
%=	operator, 89
augmented assignment, 99	//
& (ampersand)	operator, 89
operator, 90	//=
&=	augmented assignment, 99
augmented assignment, 99	/=
() (parentheses)	augmented assignment, 99
call, 86	0b
class definition, 125	integer literal, 14
function definition, 123	00
generator expression, 80	integer literal, 14
in assignment target list, 98	0x
tuple display, 78	integer literal, 14
* (asterisk)	: (colon)
function definition, 124	annotated variable, 100
import statement, 105	compound statement, 110, 111, 114, 115, 123,
in assignment target list 98	125

function annotations, 125	escape sequence, 10
in dictionary expressions, 80	\n
in formatted string literal, 12	escape sequence, 10
lambda expression, 94	\r
slicing, 86	escape sequence, 10
$:= (colon \ equals), 94$	\t
; (semicolon), 109	escape sequence, 10
< (<i>less</i>)	\U
operator, 91	escape sequence, 10
*	\u
operator, 90	escape sequence, 10
<<=	\v
augmented assignment, 99	escape sequence, 10
<=	\x
operator, 91	escape sequence, 10
!=	^ (caret)
operator, 91	operator, 90
operator, 71	^=
-=	
augmented assignment, 99	augmented assignment, 99
= (equals)	_ (underscore)
assignment statement, 97	in numeric literal, 14, 15
class definition, 43	_, identifiers, 9
for help in debugging using string	, identifiers, 9
literals, 12	abs() (object method), 51
function definition, 124	add() (object method), 50
in function calls, 86	aenter() (object method), 56
==	aexit() (object method), 56
operator, 91	aiter() (<i>object method</i>), 56
->	all (optional module attribute), 105
function annotations, 125	and() (<i>object method</i>), 50
> (greater)	anext() (<i>agen method</i>), 84
operator, 91	anext() (object method), 56
>=	annotations(<i>class attribute</i>), 28
operator, 91	annotations(function attribute), 22
>>	annotations (<i>module attribute</i>), 24, 26
operator, 90	annotations(<i>type attribute</i>), 28
>>=	await() (object method), 55
augmented assignment, 99	bases(class attribute), 28
>>>, 153	bases (type attribute), 28
@ (at)	bool() (object method), 38, 48
class definition, 126	buffer() (object method), 53
function definition, 124	bytes() (object method), 36
operator, 89	cached (<i>module attribute</i>), 24, 26
[] (square brackets)	call() (object method), 48, 88
	cause (exception attribute), 103
in assignment target list, 98	
list expression, 79	ceil() (object method), 52
subscription, 85	class (instance attribute), 29
\ (backslash)	class (method cell), 45
escape sequence, 10	class (module attribute), 39
\\	class (object attribute), 29
escape sequence, 10	class_getitem() (object class method), 46
\a	classcell (class namespace entry), 45
escape sequence, 10	closure (function attribute), 21
\b	code (function attribute), 22
escape sequence, 10	complex() (object method), 52
\f	contains() (object method), 50
escape sequence, 10	context(exception attribute), 103
/N	debug, 101

de Ferrite (function attribute) 22	in the () (chiest method) 51
defaults (function attribute), 22 del() (object method), 35	isub() (object method), 51 iter() (object method), 49
delattr() (object method), 39	itruediv() (object method), 51
delete() (object method), 40	ixor() (object method), 51
delitem() (object method), 49	kwdefaults (function attribute), 22
dict (class attribute), 28	le() (object method), 37
dict (function attribute), 22	len() (mapping object method), 38
dict (instance attribute), 29	len() (object method), 48
dict (module attribute), 27	length_hint() (object method), 48
dict(object attribute), 29	loader (module attribute), 24, 25
dict (type attribute), 28	lshift() (object method), 50
dir (module attribute), 39	lt() (object method), 37
dir() (object method), 39	main
$_$ divmod $_$ () (object method), 50	module, 60, 133
doc (class attribute), 28	matmul() (object method), 50
doc (function attribute), 22	missing() (object method), 49
doc (<i>method attribute</i>), 22, 23	mod() (<i>object method</i>), 50
doc (module attribute), 24, 26	module(class attribute), 28
doc (type attribute), 28	module(function attribute), 22
enter() (object method), 52	module(<i>method attribute</i>), 22, 23
eq() (<i>object method</i>), 37	module(<i>type attribute</i>), 28
exit() (<i>object method</i>), 52	mro (type attribute), 28
file (module attribute), 24, 26	mro_entries() (object method), 44
firstlineno(<i>class attribute</i>), 28	mul() (<i>object method</i>), 50
firstlineno (<i>type attribute</i>), 28	name (class attribute), 28
float() (object method), 52	name (function attribute), 22
floor() (object method), 52	name (<i>method attribute</i>), 22, 23
floordiv() (object method), 50	name (<i>module attribute</i>), 24, 25
format() (object method), 36	name (type attribute), 28
func(method attribute), 22, 23	ne() (object method), 37
future, 159	neg() (object method), 51
future statement, 106	new() (object method), 35
ge() (<i>object method</i>), 37	next() (generator method), 82
get() (object method), 40	objclass(<i>object attribute</i>), 41
getattr (module attribute), 39	or() (object method), 50
getattr() (<i>object method</i>), 38	package (module attribute), 24, 25
getattribute() (object method), 39	path (<i>module attribute</i>), 24, 26
getitem() (mapping object method), 34	pos() (object method), 51
getitem() (<i>object method</i>), 49	pow() (object method), 50
globals (function attribute), 21	prepare (metaclass method), 44
gt() (<i>object method</i>), 37	qualname(function attribute), 22
hash() (<i>object method</i>), 37	qualname (<i>type attribute</i>), 28
iadd() (object method), 51	radd() (<i>object method</i>), 50
iand() (<i>object method</i>), 51	rand() (<i>object method</i>), 50
ifloordiv() (object method), 51	rdivmod() (object method), 50
ilshift() (object method), 51	release_buffer() (object method), 53
imatmul() (object method), 51	repr() (object method), 36
imod() (object method), 51	reversed() (object method), 49
imul() (object method), 51	rfloordiv() (object method), 50
index() (object method), 52	rlshift() (object method), 50
init() (object method), 35	rmatmul() (<i>object method</i>), 50
init_subclass() (object class method), 42	rmod() (object method), 50
instancecheck() (type method), 46	rmul() (object method), 50
int() (object method), 52	ror() (object method), 50
invert() (object method), 51	round() (object method), 52
ior() (object method), 51	rpow() (object method), 50
ipow() (object method), 51	rrshift() (object method), 50
irshift() (object method), 51	rshift() (object method), 50
	\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

```
__rsub__() (object method), 50
                                                      import statement, 105
__rtruediv__() (object method), 50
                                                      keyword, 104, 111, 114, 115
__rxor__() (object method), 50
                                                      match statement, 115
__self__ (method attribute), 22, 23
                                                      with statement, 114
__set__() (object method), 40
                                                 AS pattern, OR pattern, capture pattern,
__set_name__() (object method), 43
                                                          wildcard pattern, 117
__setattr__() (object method), 39
                                                 ASCII, 4, 9
__setitem__() (object method), 49
                                                 asend() (agen method), 84
__slots__, 166
                                                 assert
__spec__ (module attribute), 24, 25
                                                      statement, 101
__static_attributes__(class attribute), 28
                                                 AssertionError
__static_attributes__ (type attribute), 28
                                                     exception, 101
__str__() (object method), 36
                                                 assertions
__sub___() (object method), 50
                                                      debugging, 101
__subclasscheck__() (type method), 46
                                                 assignment
__subclasses__() (type method), 29
                                                      annotated, 100
__traceback__(exception attribute), 102
                                                      attribute, 97, 98
__truediv__() (object method), 50
                                                      augmented, 99
__trunc__() (object method), 52
                                                      class attribute, 27
__type_params__(class attribute), 28
                                                      class instance attribute, 29
__type_params__(function attribute), 22
                                                      expression, 94
__type_params__ (type attribute), 28
                                                      slicing, 99
__xor__() (object method), 50
                                                      statement, 20, 97
| (vertical bar)
                                                      subscription, 99
    operator, 90
                                                      target list, 98
                                                 assignment expression, 94
    augmented assignment, 99
                                                 async
~ (tilde)
                                                      keyword, 127
    operator, 89
                                                 async def
                                                      statement, 127
Α
                                                 async for
                                                     in comprehensions, 79
                                                      statement, 127
    built-in function, 52
abstract base class, 153
                                                 async with
                                                      statement, 127
aclose() (agen method), 85
                                                 asynchronous context manager, 154
addition, 90
                                                 asynchronous generator, 154
and
                                                      asynchronous iterator, 23
    {\tt bitwise}, 90
                                                      function, 23
    operator, 94
                                                 asynchronous generator iterator, 154
annotated
                                                 asynchronous iterable, 154
    assignment, 100
                                                 asynchronous iterator, 154
annotation, 153
                                                 asynchronous-generator
annotations
                                                      object, 84
    function, 125
                                                 athrow() (agen method), 84
anonymous
                                                 atom, 77
    function, 94
                                                 attribute, 18, 154
argument, 153
                                                      assignment, 97,98
    call semantics, 86
                                                      assignment, class, 27
    function, 21
                                                      assignment, class instance, 29
    function definition, 124
                                                      class, 27
arithmetic
                                                      class instance, 29
    conversion, 77
                                                      deletion, 101
    operation, binary, 89
                                                      generic special, 18
    operation, unary, 89
                                                      reference, 85
array
                                                      special, 18
    module, 20
                                                 AttributeError
as
                                                      exception, 85
    except clause, 111
```

augmented	pow, 50, 51
assignment, 99	print, 36
await	range, 110
in comprehensions, 79	repr, 97
keyword, 88, 127	round, 52
awaitable, 154	slice, 34
	type, 17, 43
В	built-in method
b'	call, 88
	object, 24, 88
bytes literal, 10	_
b"	builtins
bytes literal, 10	module, 133
backslash character, 6	byte, 20
BDFL, 154	bytearray, 20
binary	bytecode, 29, 155
arithmetic operation, 89	bytes, 20
bitwise operation, 90	built-in function, 36
binary file, 154	bytes literal, 9
binary literal, 14	bytes-like object, 155
binding	_
global name, 107	C
name, 59, 97, 104, 105, 123, 125	c, 10
bitwise	language, 18, 19, 24, 91
and, 90	call, 86
operation, binary, 90	
operation, unary, 89	built-in function, 88
or, 90	built-in method, 88
	class instance, 88
xor, 90	class object, 27, 88
blank line, 6	function, 21, 88
block, 59	instance, 48, 88
code, 59	method, 88
BNF, 4, 77	procedure, 97
Boolean	user-defined function, 88
object, 19	callable, 155
operation, 93	object, 21, 86
borrowed reference, 154	callback, 155
break	case
statement, 104 , 110, 113	keyword, 115
built-in	match, 115
method, 24	case block, 117
built-in function	C-contiguous, 156
abs, 52	chaining
bytes, 36	comparisons, 91
call, 88	exception, 103
chr, 20	character, 20, 86
compile, 107	
complex, 52	chr
	built-in function, 20
divmod, 50, 51	class, 155
eval, 107, 134	attribute, 27
exec, 107	attribute assignment, 27
float, 52	body, 45
hash, 37	constructor, 35
id, 17	definition, 102, 125
int, 52	instance, 29
len, 1921, 48	name, 125
object, 24, 88	object, 27, 88, 125
open, 29	statement, 125
ord, 20	class instance

attribute, 29	comparisons, 37
attribute assignment, 29	chaining, 91
call, 88	compile
object, 27, 29, 88	built-in function, 107
class object	complex
call, 27, 88	built-in function, 52
class variable, 155	number, 19
clause, 109	object, 19
clear() (frame method), 33	complex literal, 14
close() (coroutine method), 56	complex number, 156
close() (generator method), 83	compound
closure variable, 155	statement, 109
co_argcount (code object attribute), 29	comprehensions, 79
co_argcount (codeobject attribute), 30	dictionary, 80
co_cellvars (code object attribute), 29	list, 7 9
co_cellvars (codeobject attribute), 30	set, 80
co_code (code object attribute), 29	Conditional
co_code (codeobject attribute), 30	expression, 93
co_consts (code object attribute), 29	conditional
co_consts (codeobject attribute), 30	expression, 94
co_filename (code object attribute), 29	constant, 9
co_filename (codeobject attribute), 30	constructor
co_firstlineno (<i>code object attribute</i>), 29	class, 35
co_firstlineno (codeobject attribute), 30	container, 17, 27
co_flags (code object attribute), 29	context, 156
co_flags (codeobject attribute), 30	context management protocol, 156
co_freevars (code object attribute), 29	context manager, 52, 156
co_freevars (codeobject attribute), 30	context variable, 156
co_kwonlyargcount (code object attribute), 29	contiguous, 156
co_kwonlyargcount (codeobject attribute), 30	continue
co_lines() (codeobject method), 31	statement, 104 , 110, 113
co_lnotab (code object attribute), 29	conversion
co_inotab (codeobject attribute), 30	arithmetic,77
co_name (code object attribute), 29	string, 36, 97
co_name (codeobject attribute), 30	coroutine, 54, 81, 156
co_names (code object attribute), 29	function, 23
co_names (codeobject attribute), 30	coroutine function, 156
co_nlocals (code object attribute), 29	CPython, 156
co_nlocals (codeobject attribute), 30	current context, 156
co_positions() (codeobject method), 31	current context, 130
co_posonlyargcount (code object attribute), 29	D
co_posonlyargcount (codeobject attribute), 30	
co_qualname (code object attribute), 29	dangling
co_qualname (codeobject attribute), 29	else, 109
co_quarname (codeobject dimbute), 30	data, 17
	type, 18
co_stacksize (codeobject attribute), 30	type, immutable, 78
co_varnames (code object attribute), 29	dbm.gnu
co_varnames (codeobject attribute), 30	module, 21
code	dbm.ndbm
block, 59	module, 21
code object, 29	debugging
collections	assertions, 101
module, 20	decimal literal, 14
comma, 78	decorator, 156
trailing, 95	DEDENT token, $7, 109$
command line, 133	def
comment, 5	statement, 123
comparison, 91	default

parameter value, 124	order, 95
definition	exc_info (in module sys), 33
class, 102, 125	except
function, 102, 123	keyword, 111
del	except_star
statement, 35, 101	keyword, 112
deletion	exception, 62, 102
attribute, 101	AssertionError, 101
target, 101	AttributeError, 85
target list, 101	chaining, 103
delimiters, 15	GeneratorExit, 83, 85
descriptor, 157	handler, 33
destructor, 35, 98	ImportError, 104
dictionary, 157	NameError, 77
comprehensions, 80	raising, 102
display, 80	StopAsyncIteration, 84
object, 21, 27, 37, 80, 85, 99	StopIteration, 82, 102
dictionary comprehension, 157	TypeError, 89
	ValueError, 90
dictionary view, 157	
display	ZeroDivisionError, 89
dictionary, 80	exception handler, 62
list, 79	exclusive
set, 80	or, 90
division, 89	exec
divmod	built-in function, 107
built-in function, 50, 51	execution
docstring, 125, 157	frame, 59, 125
documentation string, 31	restricted, 62
duck-typing, 157	stack, 33
_	execution model, 59
E	expression, 77, 157
e	assignment, 94
in numeric literal, 15	Conditional, 93
EAFP, 157	conditional, 94
elif	generator, 80
keyword, 110	lambda, 94, 125
Ellipsis	list, 95, 97
object, 18	statement, 97
else	yield, 81
conditional expression, 94	extension
dangling, 109	module, 18
keyword, 104, 110, 111, 113	extension module, 158
_	
empty list,79	F
	f'
tuple, 20, 78	-
encoding declarations (source file), 5	formatted string literal, 10 f"
environment, 60	_
environment variable	formatted string literal, 10
PYTHON_GIL, 160	f-string, 158
PYTHONHASHSEED, 38	f_back (frame attribute), 32
PYTHONNODEBUGRANGES, 31	f_builtins (frame attribute), 32
pythonpath, 72	f_code (frame attribute), 32
error handling,62	f_globals (frame attribute), 32
errors, 62	f_lasti (<i>frame attribute</i>), 32
escape sequence, 10	f_lineno (frame attribute), 32, 33
eval	f_locals (frame attribute), 32
built-in function, 107, 134	f_trace (frame attribute), 32, 33
evaluation	f_trace_lines (frame attribute), 32, 33

f_trace_opcodes (frame attribute), 32, 33	G
False, 19	garbage collection, 17, 159
file object, 158	generator, 159
file-like object, 158	expression, 80
filesystem encoding and error handler, 158	function, 23, 81, 102
finalizer, 35	iterator, 23, 102
finally	object, 31, 80, 82
keyword, 102, 104, 111, 113	generator expression, 159
find_spec	generator iterator, 159
finder, 67	GeneratorExit
finder, 67, 158	exception, 83, 85
find_spec, 67	generic
float	
built-in function, 52	special attribute, 18
floating-point	generic function, 159
number, 19	generic type, 159
object, 19	GIL, 159
floating-point literal, 14	global
floor division, 158	name binding, 107
for	namespace, 21
in comprehensions, 79	statement, 101, 107
statement, 104, 110	global interpreter lock, 159
form	grammar, 4
lambda, 94	grouping, 6
Format () (built-in function)	guard, 117
· · · · · · · · · · · · · · · · · · ·	Н
str() (object method), 36	П
formatted string literal, 12	handle an exception, 62
Fortran contiguous, 156	handler
Frame	exception, 33
execution, 59, 125	hash
object, 32	built-in function, 37
free	hash character, 5
variable, 59	hash-based pyc, 160
free threading, 158	hashable, 80, 160
free variable, 158	hexadecimal literal, 14
from	hierarchy
import statement, 59, 105	type, 18
keyword, 81, 104	hooks
yield from expression, 81	import, 67
frozenset	meta, 67
object, 21	path, 67
fstring, 12	pacii, or
f-string, 12	1
function, 158	•
annotations, 125	id
anonymous, 94	built-in function, 17
argument, 21	identifier, 8, 77
call, 21, 88	identity
call, user-defined, 88	test,93
definition, 102, 123	identity of an object, 17
generator, 81, 102	IDLE, 160
name, 123	if
object, 21, 24, 88, 123	conditional expression, 94
user-defined, 21	in comprehensions, 79
	keyword, 115
function annotation, 158	statement, 110
future	imaginary literal, 14
statement, 106	immortal, 160
	immutable, 160

data type, 78	iterable, 161
object, 20, 78, 80	unpacking, 95
immutable object, 17	iterator, 161
immutable sequence	•
object, 20	J
immutable types	j
subclassing, 35	in numeric literal, 15
import	Java
hooks, 67	language, 19
statement, 24, 104	
import hooks, 67	K
import machinery, 65	key, 80
import path, 160	key function, 161
importer, 160	key/value pair, 80
ImportError	keyword, 8, 9
exception, 104	as, 104, 111, 114, 115
importing, 160	async, 127
in	async, 127 await, 88, 127
keyword, 110	
operator, 93	case, 115
inclusive	elif, 110
or, 90	else, 104, 110, 111, 113
INDENT token, 7	except, 111
indentation, 6	except_star, 112
index operation, 19	finally, 102, 104, 111, 113
indices () (slice method), 34	from, 81, 104
inheritance, 125	if, 115
input, 134	in, 110
_	yield, 81
instance	keyword argument, 161
call, 48, 88	1
class, 29	L
object, 27, 29, 88	lambda, 161
int	expression, 94, 125
built-in function, 52	form, 94
integer, 20	language
object, 19	c, 18, 19, 24, 91
representation, 19	Java, 19
integer literal, 14	last_traceback (in module sys), 33
interactive, 160	LBYL, 162
interactive mode, 133	leading whitespace, 6
internal type, 29	len
interpolated string literal, 12	built-in function, 1921, 48
interpreted, 160	lexical analysis, 5
interpreter, 133	lexical definitions, 4
interpreter shutdown, 161	line continuation, 6
inversion, 89	line joining, 5, 6
invocation, 21	line structure, 5
io	list, 162
module, 29	assignment, target, 98
irrefutable case block, 117	comprehensions, 79
is	deletion target, 101
operator, 93	
is not	display, 79
operator, 93	empty, 79
item	expression, 95, 97
sequence, 85	object, 20, 79, 85, 86, 99
string, 86	target, 98, 110
item selection, 19	list comprehension, 162
	literal, 9, 78

loader, 67, 162	mutable object, 17
locale encoding, 162	mutable sequence
logical line,5	object, $\overset{-}{20}$
loop	-
statement, 104, 110	N
loop control	name, 8, 59, 77
target, 104	binding, 59, 97, 104, 105, 123, 125
3, -	
M	binding, global, 107
	class, 125
magic	function, 123
method, 162	mangling, 77
magic method, 162	rebinding, 97
makefile() (socket method), 29	unbinding, 101
mangling	named expression, 94
name, 77	named tuple, 163
mapping, 162	NameError
object, 21, 29, 85, 99	exception, 77
match	NameError (built-in exception), 60
case, 115	names
statement, 115	private,77
matrix multiplication, 89	namespace, 59, 163
membership	global, 21
test,93	module, 24
meta	package, 66
hooks, 67	namespace package, 163
meta hooks, 67	negation, 89
meta path finder, 162	nested scope, 163
metaclass, 43, 162	new-style class, 164
metaclass hint,44	NEWLINE token, 5, 109
method, 163	None
built-in, 24	object, 18, 97
call, 88	nonlocal
magic, 162	statement, 107
object, 22, 24, 88	not
special, 167	operator, 94
user-defined, 22	not in
method resolution order, 163	operator, 93
minus, 89	notation, 4
module, 163	NotImplemented
main, 60, 133	object, 18
array, 20	null
builtins, 133	operation, 101
collections, 20	number, 14
dbm.gnu, 21	complex, 19
dbm.ndbm, 21	floating-point, 19
extension, 18	numeric
importing, 104	object, 18, 29
io, 29	numeric literal, 14
namespace, 24	0
object, 24, 85	
sys, 112, 133	object, 17, 164
module spec, 67, 163	asynchronous-generator, 84
modulo, 89	Boolean, 19
MRO, 163	built-in function, 24, 88
mro() (type method), 28	built-in method, 24, 88
multiplication, 89	callable, 21, 86
mutable, 163	class, 27, 88, 125
object, 20, 97, 99	class instance, 27, 29, 88

code, 29	==, 91
complex, 19	> (greater), 91
dictionary, 21, 27, 37, 80, 85, 99	>=, 91
Ellipsis, 18	>>, 90
floating-point, 19	@ (at), 89
frame, 32	^ (caret), 90
frozenset, 21	(vertical bar), 90
function, 21, 24, 88, 123	~ (tilde), 89
generator, 31, 80, 82	and, 94
immutable, 20, 78, 80	in, 93
immutable sequence, 20	is, 93
instance, 27, 29, 88	is not, 93
integer, 19	not, 94
list, 20, 79, 85, 86, 99	not in,93
mapping, 21, 29, 85, 99	or, 94
method, 22, 24, 88	overloading, 34
module, 24, 85	precedence, 95
mutable, 20, 97, 99	ternary, 94
mutable sequence, 20	operators, 15
None, 18, 97	optimized scope, 164
NotImplemented, 18	or
numeric, 18, 29	bitwise, 90
sequence, 19, 29, 85, 86, 93, 99, 110	exclusive, 90
set, 21, 80	inclusive, 90
set type, 20	operator, 94
slice, 49	ord
string, 85, 86	built-in function, 20
traceback, 33, 102, 112	order
tuple, 20, 85, 86, 95	evaluation, 95
user-defined function, 21, 88, 123	output, 97
user-defined method, 22	standard, 97
bjectmatch_args(built-in variable),53	overloading
objectslots(built-in variable),41	operator, 34
octal literal, 14	_
pen	Р
built-in function, 29	package, 65, 164
peration	namespace, 66
binary arithmetic, 89	
binary bitwise, 90	portion, 66
Boolean, 93	regular, 66
	parameter, 164
null, 101	call semantics, 87
power, 88	function definition, 123
shifting, 90	value, default, 124
unary arithmetic, 89	parenthesized form, 78
unary bitwise, 89	parser, 5
perator	pass
– (minus), 89, 90	statement, 101
% (percent), 89	path
& (ampersand), 90	hooks, 67
* (asterisk), 89	path based finder, 72, 165
**, 88	path entry, 164
+ (plus), 89, 90	
/ (slash), 89	path entry finder, 165
//, 89	path entry hook, 165
< (less), 91	path hooks, 67
	path-like object, 165
<<, 90	pattern matching, 115
<=, 91	PEP, 165
!=, 91	physical line, $5, 6, 10$

. 00	106 105
plus, 89	PEP 563, 106, 125
popen () (in module os), 29	PEP 570, 125
portion, 165	PEP 572, 80, 94, 119
package, 66	PEP 585, 159
positional argument, 165	PEP 614, 124, 126
pow	PEP 617, 135
built-in function, $50,51$	PEP 626, 32
power	PEP 634, 53, 115, 123
operation, 88	PEP 636, 116, 123
precedence	PEP 646, 85, 95, 125
operator, 95	PEP 649,61
primary, 85	PEP 683, 160
print	PEP 688, 53
built-in function, 36	PEP 695, 61, 108
<pre>print() (built-in function)</pre>	PEP 696, 61, 128
str() (object method), 36	PEP 703, 158, 160
private	PEP 3104, 107
names, 77	PEP 3107, 125
procedure	PEP 3115, 45, 126
call, 97	PEP 3116, 168
program, 133	PEP 3119,46
provisional API, 165	PEP 3120,5
provisional package, 165	PEP 3129, 125, 126
Python 3000, 165	PEP 3131,8
Python Enhancement Proposals	PEP 3132, 99
PEP 1,165	PEP 3135, 45
PEP 8,92	PEP 3147, 26
PEP 236, 107	PEP 3155, 166
PEP 238, 158	PYTHON_GIL, 160
PEP 252,40	PYTHONHASHSEED, 38
PEP 255, 82	Pythonic, 165
PEP 278, 168	PYTHONNODEBUGRANGES, 31
PEP 302, 65, 75, 162	PYTHONPATH, 72
PEP 308, 94	, , <u>, , , , , , , , , , , , , , , , , </u>
PEP 318, 125, 126	Q
PEP 328, 75	qualified name, 166
PEP 338,75	qualified name, 100
PEP 342,82	R
PEP 343, 52, 115, 156	
PEP 362, 154, 164	r'
PEP 366, 25, 75	raw string literal, 10
PEP 380, 82	r"
PEP 411, 165	raw string literal, 10
PEP 414,10	raise
PEP 420, 65, 66, 71, 75, 163, 165	statement, 102
	raise an exception, 62
PEP 443, 159	raising
PEP 448, 80, 88, 95	exception, 102
PEP 451,75	range
PEP 483, 159	built-in function, 110
PEP 484, 46, 100, 125, 153, 159, 168	raw string, 10
PEP 492, 55, 82, 128, 154, 156	rebinding
PEP 498, 14, 158	name, 97
PEP 519, 165	reference
PEP 525, 82, 154	attribute, 85
PEP 526, 100, 125, 153, 168	reference count, 166
PEP 530,79	reference counting, 17
PEP 560, 44, 48	regular
PEP 562,40	package, 66
	rachage, co

regular package, 166	output, 97
relative	Standard C, 10
import, 105	standard input, 133
REPL, 166	start (slice object attribute), 34, 86
replace() (codeobject method), 32	statement, 167
repr	assert, 101
built-in function, 97	assignment, 20, 97
repr() (built-in function)	assignment, annotated, 100
repr() (object method), 36	assignment, augmented, 99
representation	async def, 127
integer, 19	async for, 127
reserved word, 8	async with, 127
restricted	break, 104 , 110, 113
execution, 62	class, 125
return	compound, 109
statement, 102 , 113	continue, 104 , 110, 113
round	def, 123
built-in function, 52	del, 35, 101
bullt-in function, 32	
S	expression, 97
	for, 104, 110
scope, 59, 60	future, 106
send() (coroutine method), 55	global, 101, 107
send() (generator method), 82	if, 110
sequence, 166	import, 24, 104
item, 85	loop, 104, 110
object, 19, 29, 85, 86, 93, 99, 110	match, 115
set	nonlocal, 107
comprehensions, 80	pass, 101
display, 80	raise, 102
object, 21, 80	return, 102 , 113
set comprehension, 167	simple, 97
set type	try, 33, 111
object, 20	type, 108
shifting	while, 104, 110
operation, 90	with, 52 , 114
simple	yield, 102
statement, 97	statement grouping, 6
single dispatch, 167	static type checker, 167
singleton	stderr (in module sys), 29
tuple, 20	stdin (in module sys), 29
slice, 86, 167	stdio, 29
built-in function, 34	stdout (in module sys), 29
object, 49	step (slice object attribute), 34, 86
slicing, 19, 20, 86	stop (slice object attribute), 34, 86
	StopAsyncIteration
assignment, 99	exception, 84
soft deprecated, 167	StopIteration
soft keyword, 9	
source character set,5	exception, 82, 102
space, 6	string
special	format() (object method), 36
attribute, 18	str() (object method), 36
attribute, generic, 18	conversion, 36, 97
method, 167	formatted literal, 12
special method, 167	immutable sequences, 20
stack	interpolated literal, 12
execution, 33	item, 86
trace, 33	object, 85, 86
standard	string literal, 9

strong reference, 167	empty, 20, 78
subclassing	object, 20, 85, 86, 95
immutable types, 35	singleton, 20
subscription, 1921, 85	type, 18, 168
assignment, 99	built-in function, 17, 43
subtraction, 90	data, 18
suite, 109	hierarchy, 18
syntax, 4	immutable data, 78
sys	statement, 108
module, 112, 133	type alias, 168
sys.exc_info,33	type hint, 168
sys.exception, 33	type of an object, 17
sys.last_traceback, 33	type parameters, 128
sys.meta_path, 67	TypeError
	exception, 89
sys.modules, 67	
sys.path, 72	types, internal, 29
sys.path_hooks,72	U
sys.path_importer_cache,72	O
sys.stderr, 29	u'
sys.stdin,29	string literal, 9
sys.stdout, 29	u"
SystemExit (built-in exception), 62	string literal, 9
т	unary
Т	arithmetic operation, 89
tab, 6	bitwise operation, 89
target,98	unbinding
deletion, 101	name, 101
list, 98, 110	UnboundLocalError, 60
list assignment, 98	Unicode, 20
list, deletion, 101	Unicode Consortium, 10
loop control, 104	universal newlines, 168
tb_frame (traceback attribute), 33, 34	UNIX, 133
tb_lasti (<i>traceback attribute</i>), 33, 34	unpacking
tb_lineno (<i>traceback attribute</i>), 33, 34	dictionary, 80
tb_next (traceback attribute), 34	in function calls, 87
termination model, 62	iterable, 95
ternary	unreachable object, 17
operator, 94	unrecognized escape sequence, 11
test	user-defined
identity, 93	function, 21
membership, 93	function call, 88
	method, 22
text encoding, 167	user-defined function
text file, 167	
throw() (coroutine method), 55	object, 21, 88, 123
throw() (generator method), 82	user-defined method
token, 5	object, 22
trace	V
stack, 33	V
traceback	value, 80
object, 33, 102, 112	default parameter, 124
trailing	value of an object, 17
comma, 95	ValueError
triple-quoted string, 167	exception, 90
triple-quoted string, 10	values
True, 19	writing, 97
try	variable
statement, 33, 111	free, 59
tuple	variable annotation, 168
	,

```
virtual environment, 168
virtual machine, 169
W
walrus operator, 94
while
    statement, 104, 110
Windows, 133
with
    \mathtt{statement}, 52, 114
writing
    values, 97
X
xor
   bitwise, 90
yield
    {\tt examples}, 83
    {\tt expression}, 81
    keyword, 81
    statement, 102
Z
Zen of Python, 169
ZeroDivisionError
    exception, 89
```