Python Setup and Usage

Release 3.13.0

Guido van Rossum and the Python development team

November 15, 2024

Python Software Foundation Email: docs@python.org

CONTENTS

1	Com	and line and environment	3
	1.1	Command line	3
		1.1.1 Interface options	3
		1.1.2 Generic options	5
		1.1.3 Miscellaneous options	6
		1.1.4 Controlling color	10
		1.1.5 Options you shouldn't use	10
	1.2	Environment variables	10
		1.2.1 Debug-mode variables	
2	Using	Python on Unix platforms	19
	2.1	Getting and installing the latest version of Python	
		2.1.1 On Linux	
		2.1.2 On FreeBSD and OpenBSD	
	2.2	Building Python	
	2.3	Python-related paths and files	
	2.4	Miscellaneous	
	2.5	Custom OpenSSL	
	2.0	opened to the second of the second opened to the se	0
3		ure Python	23
	3.1	Build Requirements	
	3.2	Generated files	
		3.2.1 configure script	
	3.3	Configure Options	
		3.3.1 General Options	
		3.3.2 C compiler options	27
		3.3.3 Linker options	27
		3.3.4 Options for third-party dependencies	27
		3.3.5 WebAssembly Options	29
		3.3.6 Install Options	29
		3.3.7 Performance options	29
		3.3.8 Python Debug Build	31
		3.3.9 Debug options	32
		3.3.10 Linker options	
		3.3.11 Libraries options	
		3.3.12 Security Options	34
		3.3.13 macOS Options	
		3.3.14 iOS Options	
		3.3.15 Cross Compiling Options	
	3.4	Python Build System	
		3.4.1 Main files of the build system	
		3.4.2 Main build steps	
		3.4.3 Main Makefile targets	
		3.4.4 C extensions	

	3.5		and linker flags	
			Preprocessor flags	
			Compiler flags	
		3.5.3	Linker flags	40
4	Heine	y Python o	on Windows	43
7	4.1	-	nstaller	43
	7,1		Installation steps	43
			Removing the MAX_PATH Limitation	44
			Installing Without UI	45
			Installing Without Downloading	47
			Modifying an install	47
		4.1.6	Installing Free-threaded Binaries	48
	4.2		rosoft Store package	48
			Known issues	49
	4.3		t.org packages	50
			Free-threaded packages	50
	4.4		eddable package	51
			Python Application	51
			Embedding Python	51
	4.5	Alternativ	ve bundles	52
	4.6	Configuri	ing Python	52
		4.6.1	Excursus: Setting environment variables	52
			Finding the Python executable	53
	4.7		node	53
	4.8	•	auncher for Windows	54
			Getting started	54
			Shebang Lines	55
			Arguments in shebang lines	56
			Customization	57
			Diagnostics	58
			Dry Run	58
			Install on demand	58
	4.9		Return codes	58
			modules	58
	4.10		al modules	60
			PyWin32	60
	4.11		cx_Freeze	60
	4.11		atforms	60
	4.12	Other I ia	auomis	OC.
5	Using	g Python o	on macOS	61
	5.1	Using Py	thon for macOS from python.org	61
		5.1.1	Installation steps	61
			How to run a Python script	69
	5.2	Alternativ	ve Distributions	69
	5.3	Installing	Additional Python Packages	69
	5.4	GUI Prog	gramming	69
	5.5	Advanced	d Topics	70
		5.5.1	Installing Free-threaded Binaries	70
		5.5.2	Installing using the command line	72
			Distributing Python Applications	73
			App Store Compliance	73
	5.6	Other Re	sources	74
6	TT~!	v Dv-4la	on Android	7-
6			on Android	75 75
	6.1	Adding P	Python to an Android app	75
7	Using	g Python o	on iOS	77

	7.1	7.1.1 7.1.2	at runtime on iOS	77 77
		7.1.3	Standard library availability	
		7.1.4 7.1.5	Binary extension modules	
	7.2		g Python on iOS	
	1.2	7.2.1	Tools for building iOS apps	
		7.2.1	Adding Python to an iOS project	
	7.3		ore Compliance	
_			•	
8	Edito	ors and I	DEs	83
A	Gloss	sary		85
В	Abou	ıt these d	locuments	103
	B.1		utors to the Python Documentation	
\mathbf{C}	Histo	ry and I		105
	C.1		of the software	
	C.2	Terms a	and conditions for accessing or otherwise using Python	
		C.2.1	PSF LICENSE AGREEMENT FOR PYTHON 3.13.0	
		C.2.2	BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0	
		C.2.3	CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1	
		C.2.4	CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2	108
		C.2.5	ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMEN-	
			TATION	109
	C.3	License	s and Acknowledgements for Incorporated Software	109
		C.3.1	Mersenne Twister	
		C.3.2	Sockets	
		C.3.3	Asynchronous socket services	
		C.3.4	Cookie management	111
		C.3.5	Execution tracing	
		C.3.6	UUencode and UUdecode functions	112
		C.3.7	XML Remote Procedure Calls	113
		C.3.8	test_epoll	113
		C.3.9	Select kqueue	114
		C.3.10	SipHash24	114
		C.3.11	strtod and dtoa	115
		C.3.12	OpenSSL	115
		C.3.13	expat	118
		C.3.14	libffi	119
		C.3.15	zlib	119
		C.3.16	cfuhash	120
		C.3.17	libmpdec	120
		C.3.18	W3C C14N test suite	121
		C.3.19	mimalloc	
		C.3.20	asyncio	
		C.3.21	Global Unbounded Sequences (GUS)	122
D	Сору	right		125
	dex	J		127
111	ula			14/

This part of the documentation is devoted to general information on the setup of the Python environment on different platforms, the invocation of the interpreter and things that make working with Python easier.

CONTENTS 1

2 CONTENTS

COMMAND LINE AND ENVIRONMENT

The CPython interpreter scans the command line and the environment for various settings.

CPython implementation detail: Other implementations' command line schemes may differ. See implementations for further resources.

1.1 Command line

When invoking Python, you may specify any of these options:

```
python [-bBdEhiIOPqRsSuvVWx?] [-c command | -m module-name | script | - ] [args]
```

The most common use case is, of course, a simple invocation of a script:

python myscript.py

1.1.1 Interface options

The interpreter interface resembles that of the UNIX shell, but provides some additional methods of invocation:

- When called with standard input connected to a tty device, it prompts for commands and executes them until an EOF (an end-of-file character, you can produce that with Ctrl-D on UNIX or Ctrl-Z, Enter on Windows) is read. For more on interactive mode, see tut-interac.
- When called with a file name argument or with a file as standard input, it reads and executes a script from that file.
- When called with a directory name argument, it reads and executes an appropriately named script from that directory.
- When called with -c command, it executes the Python statement(s) given as *command*. Here *command* may contain multiple statements separated by newlines. Leading whitespace is significant in Python statements!
- When called with -m module-name, the given module is located on the Python module path and executed as a script.

In non-interactive mode, the entire input is parsed before it is executed.

An interface option terminates the list of options consumed by the interpreter, all consecutive arguments will end up in sys.argv - note that the first element, subscript zero (sys.argv[0]), is a string reflecting the program's source.

-c <command>

Execute the Python code in *command*. *command* can be one or more statements separated by newlines, with significant leading whitespace as in normal module code.

If this option is given, the first element of sys.argv will be "-c" and the current directory will be added to the start of sys.path (allowing modules in that directory to be imported as top level modules).

Raises an auditing event cpython.run_command with argument command.

-m <module-name>

Search sys.path for the named module and execute its contents as the __main__ module.

Since the argument is a *module* name, you must not give a file extension (.py). The module name should be a valid absolute Python module name, but the implementation may not always enforce this (e.g. it may allow you to use a name that includes a hyphen).

Package names (including namespace packages) are also permitted. When a package name is supplied instead of a normal module, the interpreter will execute <pkg>.__main__ as the main module. This behaviour is deliberately similar to the handling of directories and zipfiles that are passed to the interpreter as the script argument.



This option cannot be used with built-in modules and extension modules written in C, since they do not have Python module files. However, it can still be used for precompiled modules, even if the original source file is not available.

If this option is given, the first element of sys.argv will be the full path to the module file (while the module file is being located, the first element will be set to "-m"). As with the -c option, the current directory will be added to the start of sys.path.

-I option can be used to run the script in isolated mode where sys.path contains neither the current directory nor the user's site-packages directory. All PYTHON* environment variables are ignored, too.

Many standard library modules contain code that is invoked on their execution as a script. An example is the timeit module:

```
python -m timeit -s "setup here" "benchmarked code here"
python -m timeit -h # for details
```

Raises an auditing event cpython.run_module with argument module-name.

→ See also

runpy.run_module()

Equivalent functionality directly available to Python code

PEP 338 - Executing modules as scripts

Changed in version 3.1: Supply the package name to run a __main__ submodule.

Changed in version 3.4: namespace packages are also supported

Read commands from standard input (sys.stdin). If standard input is a terminal, -i is implied.

If this option is given, the first element of sys.argv will be "-" and the current directory will be added to the start of sys.path.

Raises an auditing event cpython.run_stdin with no arguments.

<script>

Execute the Python code contained in *script*, which must be a filesystem path (absolute or relative) referring to either a Python file, a directory containing a __main__.py file, or a zipfile containing a __main__.py file.

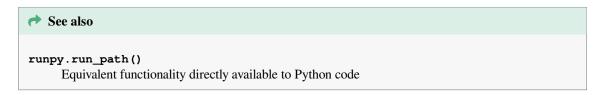
If this option is given, the first element of sys.argv will be the script name as given on the command line.

If the script name refers directly to a Python file, the directory containing that file is added to the start of sys.path, and the file is executed as the __main__ module.

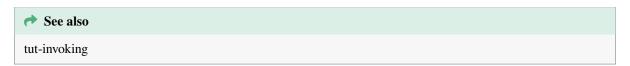
If the script name refers to a directory or zipfile, the script name is added to the start of sys.path and the __main__.py file in that location is executed as the __main__ module.

-I option can be used to run the script in isolated mode where sys.path contains neither the script's directory nor the user's site-packages directory. All PYTHON* environment variables are ignored, too.

Raises an auditing event cpython.run_file with argument filename.



If no interface option is given, -i is implied, sys.argv[0] is an empty string ("") and the current directory will be added to the start of sys.path. Also, tab-completion and history editing is automatically enabled, if available on your platform (see rlcompleter-config).



Changed in version 3.4: Automatic enabling of tab-completion and history editing.

1.1.2 Generic options

-?

-h

--help

Print a short description of all command line options and corresponding environment variables and exit.

--help-env

Print a short description of Python-specific environment variables and exit.

Added in version 3.11.

--help-xoptions

Print a description of implementation-specific –*X* options and exit.

Added in version 3.11.

--help-all

Print complete usage information and exit.

Added in version 3.11.

-v

--version

Print the Python version number and exit. Example output could be:

```
Python 3.8.0b2+
```

When given twice, print more information about the build, like:

```
Python 3.8.0b2+ (3.8:0c076caaa8, Apr 20 2019, 21:55:00)
[GCC 6.2.0 20161005]
```

Added in version 3.6: The -VV option.

1.1. Command line 5

1.1.3 Miscellaneous options

-b

Issue a warning when converting bytes or bytearray to str without specifying encoding or comparing bytes or bytearray with str or bytes with int. Issue an error when the option is given twice (-bb).

Changed in version 3.5: Affects also comparisons of bytes with int.

-B

If given, Python won't try to write .pyc files on the import of source modules. See also PYTHONDONTWRITEBYTECODE.

--check-hash-based-pycs default|always|never

Control the validation behavior of hash-based .pyc files. See pyc-invalidation. When set to default, checked and unchecked hash-based bytecode cache files are validated according to their default semantics. When set to always, all hash-based .pyc files, whether checked or unchecked, are validated against their corresponding source file. When set to never, hash-based .pyc files are not validated against their corresponding source files.

The semantics of timestamp-based .pyc files are unaffected by this option.

-d

Turn on parser debugging output (for expert only). See also the PYTHONDEBUG environment variable.

This option requires a *debug build of Python*, otherwise it's ignored.

 $-\mathbf{E}$

Ignore all PYTHON* environment variables, e.g. PYTHONPATH and PYTHONHOME, that might be set.

See also the -P and -I (isolated) options.

-i

When a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command, even when sys.stdin does not appear to be a terminal. The PYTHONSTARTUP file is not read

This can be useful to inspect global variables or a stack trace when a script raises an exception. See also <code>PYTHONINSPECT</code>.

-I

Run Python in isolated mode. This also implies -E, -P and -s options.

In isolated mode sys.path contains neither the script's directory nor the user's site-packages directory. All PYTHON* environment variables are ignored, too. Further restrictions may be imposed to prevent the user from injecting malicious code.

Added in version 3.4.

-0

Remove assert statements and any code conditional on the value of __debug__. Augment the filename for compiled (*bytecode*) files by adding .opt-1 before the .pyc extension (see PEP 488). See also PYTHONOPTIMIZE.

Changed in version 3.5: Modify .pyc filenames according to PEP 488.

-00

Do -0 and also discard docstrings. Augment the filename for compiled (*bytecode*) files by adding .opt-2 before the .pyc extension (see **PEP 488**).

Changed in version 3.5: Modify .pyc filenames according to PEP 488.

-P

Don't prepend a potentially unsafe path to sys.path:

• python -m module command line: Don't prepend the current working directory.

- python script.py command line: Don't prepend the script's directory. If it's a symbolic link, resolve symbolic links.
- python -c code and python (REPL) command lines: Don't prepend an empty string, which means the current working directory.

See also the PYTHONSAFEPATH environment variable, and -E and -I (isolated) options.

Added in version 3.11.

-q

Don't display the copyright and version messages even in interactive mode.

Added in version 3.2.

-R

Turn on hash randomization. This option only has an effect if the PYTHONHASHSEED environment variable is set to 0, since hash randomization is enabled by default.

On previous versions of Python, this option turns on hash randomization, so that 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.

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

PYTHONHASHSEED allows you to set a fixed value for the hash seed secret.

Added in version 3.2.3.

Changed in version 3.7: The option is no longer ignored.

-s

Don't add the user site-packages directory to sys.path.

See also PYTHONNOUSERSITE.



PEP 370 - Per user site-packages directory

-s

Disable the import of the module site and the site-dependent manipulations of sys.path that it entails. Also disable these manipulations if site is explicitly imported later (call site.main() if you want them to be triggered).

-u

Force the stdout and stderr streams to be unbuffered. This option has no effect on the stdin stream.

See also PYTHONUNBUFFERED.

Changed in version 3.7: The text layer of the stdout and stderr streams now is unbuffered.

-v

Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. When given twice (-vv), print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Changed in version 3.10: The site module reports the site-specific paths and .pth files being processed.

See also PYTHONVERBOSE.

1.1. Command line 7

-W arg

Warning control. Python's warning machinery by default prints warning messages to sys.stderr.

The simplest settings apply a particular action unconditionally to all warnings emitted by a process (even those that are otherwise ignored by default):

```
-Wdefault # Warn once per call location
-Werror # Convert to exceptions
-Walways # Warn every time
-Wall # Same as -Walways
-Wmodule # Warn once per calling module
-Wonce # Warn once per Python process
-Wignore # Never warn
```

The action names can be abbreviated as desired and the interpreter will resolve them to the appropriate action name. For example, -Wi is the same as -Wignore.

The full form of argument is:

```
action:message:category:module:lineno
```

Empty fields match all values; trailing empty fields may be omitted. For example -W ignore::DeprecationWarning ignores all DeprecationWarning warnings.

The action field is as explained above but only applies to warnings that match the remaining fields.

The *message* field must match the whole warning message; this match is case-insensitive.

The *category* field matches the warning category (ex: DeprecationWarning). This must be a class name; the match test whether the actual warning category of the message is a subclass of the specified warning category.

The *module* field matches the (fully qualified) module name; this match is case-sensitive.

The *lineno* field matches the line number, where zero matches all line numbers and is thus equivalent to an omitted line number.

Multiple $-\overline{W}$ options can be given; when a warning matches more than one option, the action for the last matching option is performed. Invalid $-\overline{W}$ options are ignored (though, a warning message is printed about invalid options when the first warning is issued).

Warnings can also be controlled using the PYTHONWARNINGS environment variable and from within a Python program using the warnings module. For example, the warnings filterwarnings () function can be used to use a regular expression on the warning message.

See warning-filter and describing-warning-filters for more details.

-x

Skip the first line of the source, allowing use of non-Unix forms of #! cmd. This is intended for a DOS specific hack only.

-x

Reserved for various implementation-specific options. CPython currently defines the following possible values:

• -X faulthandler to enable faulthandler. See also PYTHONFAULTHANDLER.

Added in version 3.3.

- -X showrefcount to output the total reference count and number of used memory blocks when the program finishes or after each statement in the interactive interpreter. This only works on *debug builds*. Added in version 3.4.
- —X tracemalloc to start tracing Python memory allocations using the tracemalloc module. By default, only the most recent frame is stored in a traceback of a trace. Use —X tracemalloc=NFRAME to start tracing with a traceback limit of NFRAME frames. See tracemalloc.start() and PYTHONTRACEMALLOC for more information.

Added in version 3.4.

 -X int_max_str_digits configures the integer string conversion length limitation. See also PYTHONINTMAXSTRDIGITS.

Added in version 3.11.

• —X importtime to show how long each import takes. It shows module name, cumulative time (including nested imports) and self time (excluding nested imports). Note that its output may be broken in multithreaded application. Typical usage is python3 —X importtime —c 'import asyncio'. See also PYTHONPROFILEIMPORTTIME.

Added in version 3.7.

• -X dev: enable Python Development Mode, introducing additional runtime checks that are too expensive to be enabled by default. See also PYTHONDEVMODE.

Added in version 3.7.

• -X utf8 enables the Python UTF-8 Mode. -X utf8=0 explicitly disables Python UTF-8 Mode (even when it would otherwise activate automatically). See also PYTHONUTF8.

Added in version 3.7.

-X pycache_prefix=PATH enables writing .pyc files to a parallel tree rooted at the given directory instead of to the code tree. See also PYTHONPYCACHEPREFIX.

Added in version 3.8.

• -X warn_default_encoding issues a EncodingWarning when the locale-specific default encoding is used for opening files. See also PYTHONWARNDEFAULTENCODING.

Added in version 3.10.

• -x no_debug_ranges disables the inclusion of the tables mapping extra location information (end line, start column offset and end column offset) to every instruction in code objects. This is useful when smaller code objects and pyc files are desired as well as suppressing the extra visual location indicators when the interpreter displays tracebacks. See also PYTHONNODEBUGRANGES.

Added in version 3.11.

• -X frozen_modules determines whether or not frozen modules are ignored by the import machinery. A value of on means they get imported and off means they are ignored. The default is on if this is an installed Python (the normal case). If it's under development (running from the source tree) then the default is off. Note that the importlib_bootstrap and importlib_bootstrap_external frozen modules are always used, even if this flag is set to off. See also PYTHON_FROZEN_MODULES.

Added in version 3.11.

• -X perf enables support for the Linux perf profiler. When this option is provided, the perf profiler will be able to report Python calls. This option is only available on some platforms and will do nothing if is not supported on the current system. The default value is "off". See also PYTHONPERFSUPPORT and perf_profiling.

Added in version 3.12.

• -X perf_jit enables support for the Linux perf profiler with DWARF support. When this option is provided, the perf profiler will be able to report Python calls using DWARF information. This option is only available on some platforms and will do nothing if is not supported on the current system. The default value is "off". See also PYTHON_PERF_JIT_SUPPORT and perf_profiling.

Added in version 3.13.

• -X cpu_count=n overrides os.cpu_count(), os.process_cpu_count(), and multiprocessing.cpu_count(). n must be greater than or equal to 1. This option may be useful for users who need to limit CPU resources of a container system. See also PYTHON_CPU_COUNT. If n is default, nothing is overridden.

Added in version 3.13.

1.1. Command line 9

• -X presite=package.module specifies a module that should be imported before the site module is executed and before the __main__ module exists. Therefore, the imported module isn't __main__. This can be used to execute code early during Python initialization. Python needs to be built in debug mode for this option to exist. See also PYTHON_PRESITE.

Added in version 3.13.

• -X gil=0, 1 forces the GIL to be disabled or enabled, respectively. Setting to 0 is only available in builds configured with --disable-gil. See also PYTHON_GIL and whatsnew313-free-threaded-cpython.

Added in version 3.13.

It also allows passing arbitrary values and retrieving them through the sys._xoptions dictionary.

Added in version 3.2.

Changed in version 3.9: Removed the -X showalloccount option.

Changed in version 3.10: Removed the -X oldparser option.

1.1.4 Controlling color

The Python interpreter is configured by default to use colors to highlight output in certain situations such as when displaying tracebacks. This behavior can be controlled by setting different environment variables.

Setting the environment variable TERM to dumb will disable color.

If the FORCE_COLOR environment variable is set, then color will be enabled regardless of the value of TERM. This is useful on CI systems which aren't terminals but can still display ANSI escape sequences.

If the NO_COLOR environment variable is set, Python will disable all color in the output. This takes precedence over FORCE_COLOR.

All these environment variables are used also by other tools to control color output. To control the color output only in the Python interpreter, the <code>PYTHON_COLORS</code> environment variable can be used. This variable takes precedence over <code>NO_COLOR</code>, which in turn takes precedence over <code>FORCE_COLOR</code>.

1.1.5 Options you shouldn't use

-J

Reserved for use by Jython.

1.2 Environment variables

These environment variables influence Python's behavior, they are processed before the command-line switches other than -E or -I. It is customary that command-line switches override environmental variables where there is a conflict.

PYTHONHOME

Change the location of the standard Python libraries. By default, the libraries are searched in <code>prefix/lib/pythonversion</code> and <code>exec_prefix/lib/pythonversion</code>, where <code>prefix</code> and <code>exec_prefix</code> are installation-dependent directories, both defaulting to <code>/usr/local</code>.

When PYTHONHOME is set to a single directory, its value replaces both prefix and exec_prefix. To specify different values for these, set PYTHONHOME to prefix: exec_prefix.

PYTHONPATH

Augment the default search path for module files. The format is the same as the shell's PATH: one or more directory pathnames separated by os.pathsep (e.g. colons on Unix or semicolons on Windows). Non-existent directories are silently ignored.

In addition to normal directories, individual *PYTHONPATH* entries may refer to zipfiles containing pure Python modules (in either source or compiled form). Extension modules cannot be imported from zipfiles.

The default search path is installation dependent, but generally begins with prefix/lib/pythonversion (see PYTHONHOME above). It is always appended to PYTHONPATH.

An additional directory will be inserted in the search path in front of <code>PYTHONPATH</code> as described above under <code>Interface options</code>. The search path can be manipulated from within a Python program as the variable <code>sys.path</code>.

PYTHONSAFEPATH

If this is set to a non-empty string, don't prepend a potentially unsafe path to sys.path: see the -P option for details.

Added in version 3.11.

PYTHONPLATLIBDIR

If this is set to a non-empty string, it overrides the sys.platlibdir value.

Added in version 3.9.

PYTHONSTARTUP

If this is the name of a readable file, the Python commands in that file are executed before the first prompt is displayed in interactive mode. The file is executed in the same namespace where interactive commands are executed so that objects defined or imported in it can be used without qualification in the interactive session. You can also change the prompts <code>sys.ps1</code> and <code>sys.ps2</code> and the hook <code>sys.__interactivehook__</code> in this file.

Raises an auditing event cpython.run_startup with the filename as the argument when called on startup.

PYTHONOPTIMIZE

If this is set to a non-empty string it is equivalent to specifying the -0 option. If set to an integer, it is equivalent to specifying -0 multiple times.

PYTHONBREAKPOINT

If this is set, it names a callable using dotted-path notation. The module containing the callable will be imported and then the callable will be run by the default implementation of <code>sys.breakpointhook()</code> which itself is called by built-in <code>breakpoint()</code>. If not set, or set to the empty string, it is equivalent to the value "pdb.set_trace". Setting this to the string "0" causes the default implementation of <code>sys.breakpointhook()</code> to do nothing but return immediately.

Added in version 3.7.

PYTHONDEBUG

If this is set to a non-empty string it is equivalent to specifying the -d option. If set to an integer, it is equivalent to specifying -d multiple times.

This environment variable requires a *debug build of Python*, otherwise it's ignored.

PYTHONINSPECT

If this is set to a non-empty string it is equivalent to specifying the -i option.

This variable can also be modified by Python code using os.environ to force inspect mode on program termination.

Raises an auditing event cpython.run_stdin with no arguments.

Changed in version 3.12.5: (also 3.11.10, 3.10.15, 3.9.20, and 3.8.20) Emits audit events.

Changed in version 3.13: Uses PyREPL if possible, in which case PYTHONSTARTUP is also executed. Emits audit events.

PYTHONUNBUFFERED

If this is set to a non-empty string it is equivalent to specifying the -u option.

PYTHONVERBOSE

If this is set to a non-empty string it is equivalent to specifying the -v option. If set to an integer, it is equivalent to specifying -v multiple times.

PYTHONCASEOK

If this is set, Python ignores case in import statements. This only works on Windows and macOS.

PYTHONDONTWRITEBYTECODE

If this is set to a non-empty string, Python won't try to write .pyc files on the import of source modules. This is equivalent to specifying the -B option.

PYTHONPYCACHEPREFIX

If this is set, Python will write .pyc files in a mirror directory tree at this path, instead of in __pycache_directories within the source tree. This is equivalent to specifying the -X pycache_prefix=PATH option.

Added in version 3.8.

PYTHONHASHSEED

If this variable is not set or set to random, a random value is used to seed the hashes of str and bytes objects.

If PYTHONHASHSEED is set to an integer value, it is used as a fixed seed for generating the hash() of the types covered by the hash randomization.

Its purpose is to allow repeatable hashing, such as for selftests for the interpreter itself, or to allow a cluster of python processes to share hash values.

The integer must be a decimal number in the range [0,4294967295]. Specifying the value 0 will disable hash randomization.

Added in version 3.2.3.

PYTHONINTMAXSTRDIGITS

If this variable is set to an integer, it is used to configure the interpreter's global integer string conversion length limitation.

Added in version 3.11.

PYTHONIOENCODING

If this is set before running the interpreter, it overrides the encoding used for stdin/stdout/stderr, in the syntax encodingname:errorhandler. Both the encodingname and the:errorhandler parts are optional and have the same meaning as in str.encode().

For stderr, the :errorhandler part is ignored; the handler will always be 'backslashreplace'.

Changed in version 3.4: The encodingname part is now optional.

Changed in version 3.6: On Windows, the encoding specified by this variable is ignored for interactive console buffers unless <code>PYTHONLEGACYWINDOWSSTDIO</code> is also specified. Files and pipes redirected through the standard streams are not affected.

PYTHONNOUSERSITE

If this is set, Python won't add the user site-packages directory to sys.path.



PEP 370 – Per user site-packages directory

PYTHONUSERBASE

Defines the user base directory, which is used to compute the path of the user site-packages directory and installation paths for python -m pip install --user.



PEP 370 - Per user site-packages directory

PYTHONEXECUTABLE

If this environment variable is set, sys.argv[0] will be set to its value instead of the value got through the C runtime. Only works on macOS.

PYTHONWARNINGS

This is equivalent to the -W option. If set to a comma separated string, it is equivalent to specifying -W multiple times, with filters later in the list taking precedence over those earlier in the list.

The simplest settings apply a particular action unconditionally to all warnings emitted by a process (even those that are otherwise ignored by default):

```
PYTHONWARNINGS=default # Warn once per call location
PYTHONWARNINGS=error # Convert to exceptions
PYTHONWARNINGS=always # Warn every time
PYTHONWARNINGS=all # Same as PYTHONWARNINGS=always
PYTHONWARNINGS=module # Warn once per calling module
PYTHONWARNINGS=once # Warn once per Python process
PYTHONWARNINGS=ignore # Never warn
```

See warning-filter and describing-warning-filters for more details.

PYTHONFAULTHANDLER

If this environment variable is set to a non-empty string, faulthandler.enable() is called at startup: install a handler for SIGSEGV, SIGFPE, SIGABRT, SIGBUS and SIGILL signals to dump the Python traceback. This is equivalent to -X faulthandler option.

Added in version 3.3.

PYTHONTRACEMALLOC

If this environment variable is set to a non-empty string, start tracing Python memory allocations using the tracemalloc module. The value of the variable is the maximum number of frames stored in a traceback of a trace. For example, PYTHONTRACEMALLOC=1 stores only the most recent frame. See the tracemalloc. start () function for more information. This is equivalent to setting the -X tracemalloc option.

Added in version 3.4.

PYTHONPROFILEIMPORTTIME

If this environment variable is set to a non-empty string, Python will show how long each import takes. This is equivalent to setting the -X importtime option.

Added in version 3.7.

PYTHONASYNCIODEBUG

If this environment variable is set to a non-empty string, enable the debug mode of the asyncio module.

Added in version 3.4.

PYTHONMALLOC

Set the Python memory allocators and/or install debug hooks.

Set the family of memory allocators used by Python:

- default: use the default memory allocators.
- malloc: use the malloc() function of the C library for all domains (PYMEM_DOMAIN_RAW, PYMEM_DOMAIN_MEM, PYMEM_DOMAIN_OBJ).
- pymalloc: use the pymalloc allocator for PYMEM_DOMAIN_MEM and PYMEM_DOMAIN_OBJ domains and use the malloc() function for the PYMEM_DOMAIN_RAW domain.
- mimalloc: use the mimalloc allocator for PYMEM_DOMAIN_MEM and PYMEM_DOMAIN_OBJ domains and use the malloc() function for the PYMEM_DOMAIN_RAW domain.

Install debug hooks:

• debug: install debug hooks on top of the default memory allocators.

- malloc_debug: same as malloc but also install debug hooks.
- pymalloc_debug: same as pymalloc but also install debug hooks.
- mimalloc_debug: same as mimalloc but also install debug hooks.

Added in version 3.6.

Changed in version 3.7: Added the "default" allocator.

PYTHONMALLOCSTATS

If set to a non-empty string, Python will print statistics of the pymalloc memory allocator every time a new pymalloc object arena is created, and on shutdown.

This variable is ignored if the PYTHONMALLOC environment variable is used to force the malloc() allocator of the C library, or if Python is configured without pymalloc support.

Changed in version 3.6: This variable can now also be used on Python compiled in release mode. It now has no effect if set to an empty string.

PYTHONLEGACYWINDOWSFSENCODING

If set to a non-empty string, the default *filesystem encoding and error handler* mode will revert to their pre-3.6 values of 'mbcs' and 'replace', respectively. Otherwise, the new defaults 'utf-8' and 'surrogatepass' are used.

This may also be enabled at runtime with sys._enablelegacywindowsfsencoding().

Availability: Windows.

Added in version 3.6: See PEP 529 for more details.

PYTHONLEGACYWINDOWSSTDIO

If set to a non-empty string, does not use the new console reader and writer. This means that Unicode characters will be encoded according to the active console code page, rather than using utf-8.

This variable is ignored if the standard streams are redirected (to files or pipes) rather than referring to console buffers.

Availability: Windows.

Added in version 3.6.

PYTHONCOERCECLOCALE

If set to the value 0, causes the main Python command line application to skip coercing the legacy ASCII-based C and POSIX locales to a more capable UTF-8 based alternative.

If this variable is *not* set (or is set to a value other than 0), the LC_ALL locale override environment variable is also not set, and the current locale reported for the LC_CTYPE category is either the default C locale, or else the explicitly ASCII-based POSIX locale, then the Python CLI will attempt to configure the following locales for the LC_CTYPE category in the order listed before loading the interpreter runtime:

- C.UTF-8
- C.utf8
- UTF-8

If setting one of these locale categories succeeds, then the LC_CTYPE environment variable will also be set accordingly in the current process environment before the Python runtime is initialized. This ensures that in addition to being seen by both the interpreter itself and other locale-aware components running in the same process (such as the GNU readline library), the updated setting is also seen in subprocesses (regardless of whether or not those processes are running a Python interpreter), as well as in operations that query the environment rather than the current C locale (such as Python's own locale.getdefaultlocale()).

Configuring one of these locales (either explicitly or via the above implicit locale coercion) automatically enables the surrogateescape error handler for sys.stdin and sys.stdout (sys.stderr continues to use backslashreplace as it does in any other locale). This stream handling behavior can be overridden using PYTHONIOENCODING as usual.

For debugging purposes, setting PYTHONCOERCECLOCALE=warn will cause Python to emit warning messages on stderr if either the locale coercion activates, or else if a locale that *would* have triggered coercion is still active when the Python runtime is initialized.

Also note that even when locale coercion is disabled, or when it fails to find a suitable target locale, *PYTHONUTF8* will still activate by default in legacy ASCII-based locales. Both features must be disabled in order to force the interpreter to use ASCII instead of UTF-8 for system interfaces.

Availability: Unix.

Added in version 3.7: See PEP 538 for more details.

PYTHONDEVMODE

If this environment variable is set to a non-empty string, enable Python Development Mode, introducing additional runtime checks that are too expensive to be enabled by default. This is equivalent to setting the -X dev option.

Added in version 3.7.

PYTHONUTF8

If set to 1, enable the Python UTF-8 Mode.

If set to 0, disable the Python UTF-8 Mode.

Setting any other non-empty string causes an error during interpreter initialisation.

Added in version 3.7.

PYTHONWARNDEFAULTENCODING

If this environment variable is set to a non-empty string, issue a EncodingWarning when the locale-specific default encoding is used.

See io-encoding-warning for details.

Added in version 3.10.

PYTHONNODEBUGRANGES

If this variable is set, it disables the inclusion of the tables mapping extra location information (end line, start column offset and end column offset) to every instruction in code objects. This is useful when smaller code objects and pyc files are desired as well as suppressing the extra visual location indicators when the interpreter displays tracebacks.

Added in version 3.11.

PYTHONPERFSUPPORT

If this variable is set to a nonzero value, it enables support for the Linux perf profiler so Python calls can be detected by it.

If set to 0, disable Linux perf profiler support.

See also the -X perf command-line option and perf_profiling.

Added in version 3.12.

PYTHON PERF JIT SUPPORT

If this variable is set to a nonzero value, it enables support for the Linux perf profiler so Python calls can be detected by it using DWARF information.

If set to 0, disable Linux perf profiler support.

See also the -X perf_jit command-line option and perf_profiling.

Added in version 3.13.

PYTHON_CPU_COUNT

If this variable is set to a positive integer, it overrides the return values of $os.cpu_count()$ and $os.process_cpu_count()$.

See also the -X cpu_count command-line option.

Added in version 3.13.

PYTHON_FROZEN_MODULES

If this variable is set to on or off, it determines whether or not frozen modules are ignored by the import machinery. A value of on means they get imported and off means they are ignored. The default is on for non-debug builds (the normal case) and off for debug builds. Note that the importlib_bootstrap and importlib_bootstrap_external frozen modules are always used, even if this flag is set to off.

See also the -X frozen_modules command-line option.

Added in version 3.13.

PYTHON COLORS

If this variable is set to 1, the interpreter will colorize various kinds of output. Setting it to 0 deactivates this behavior. See also *Controlling color*.

Added in version 3.13.

PYTHON_BASIC_REPL

If this variable is set to 1, the interpreter will not attempt to load the Python-based *REPL* that requires curses and readline, and will instead use the traditional parser-based *REPL*.

Added in version 3.13.

PYTHON_HISTORY

This environment variable can be used to set the location of a .python_history file (by default, it is .python_history in the user's home directory).

Added in version 3.13.

PYTHON_GIL

If this variable is set to 1, the global interpreter lock (GIL) will be forced on. Setting it to 0 forces the GIL off (needs Python configured with the -disable-gil build option).

See also the -X gil command-line option, which takes precedence over this variable, and whatsnew313-free-threaded-cpython.

Added in version 3.13.

1.2.1 Debug-mode variables

PYTHONDUMPREFS

If set, Python will dump objects and reference counts still alive after shutting down the interpreter.

Needs Python configured with the --with-trace-refs build option.

PYTHONDUMPREFSFILE

If set, Python will dump objects and reference counts still alive after shutting down the interpreter into a file under the path given as the value to this environment variable.

Needs Python configured with the --with-trace-refs build option.

Added in version 3.11.

PYTHON PRESITE

If this variable is set to a module, that module will be imported early in the interpreter lifecycle, before the site module is executed, and before the __main__ module is created. Therefore, the imported module is not treated as __main__.

This can be used to execute code early during Python initialization.

To import a submodule, use package.module as the value, like in an import statement.

See also the -X presite command-line option, which takes precedence over this variable.

Needs Python configured with the --with-pydebug build option.

Added in version 3.13.

USING PYTHON ON UNIX PLATFORMS

2.1 Getting and installing the latest version of Python

2.1.1 On Linux

Python comes preinstalled on most Linux distributions, and is available as a package on all others. However there are certain features you might want to use that are not available on your distro's package. You can compile the latest version of Python from source.

In the event that the latest version of Python doesn't come preinstalled and isn't in the repositories as well, you can make packages for your own distro. Have a look at the following links:

♂ See also

https://www.debian.org/doc/manuals/maint-guide/first.en.html

for Debian users

https://en.opensuse.org/Portal:Packaging

for OpenSuse users

https://docs.fedoraproject.org/en-US/package-maintainers/Packaging_Tutorial_GNU_Hello/ for Fedora users

https://slackbook.org/html/package-management-making-packages.html

for Slackware users

2.1.2 On FreeBSD and OpenBSD

• FreeBSD users, to add the package use:

```
pkg install python3
```

• OpenBSD users, to add the package use:

```
pkg_add -r python

pkg_add ftp://ftp.openbsd.org/pub/OpenBSD/4.2/packages/<insert your_
→architecture here>/python-<version>.tgz
```

For example i386 users get the 2.5.1 version of Python using:

pkg_add ftp://ftp.openbsd.org/pub/OpenBSD/4.2/packages/i386/python-2.5.1p2.tqz

2.2 Building Python

If you want to compile CPython yourself, first thing you should do is get the source. You can download either the latest release's source or just grab a fresh clone. (If you want to contribute patches, you will need a clone.)

The build process consists of the usual commands:

```
./configure
make
make install
```

Configuration options and caveats for specific Unix platforms are extensively documented in the README.rst file in the root of the Python source tree.

⚠ Warning

make install can overwrite or masquerade the python3 binary. make altinstall is therefore recommended instead of make install since it only installs <code>exec_prefix/bin/pythonversion</code>.

2.3 Python-related paths and files

These are subject to difference depending on local installation conventions; prefix and exec_prefix are installation-dependent and should be interpreted as for GNU software; they may be the same.

For example, on most Linux systems, the default for both is /usr.

File/directory	Meaning
exec_prefix/bin/python3	Recommended location of the interpreter.
prefix/lib/pythonversion,	Recommended locations of the directories containing the standard
<pre>exec_prefix/lib/pythonversion</pre>	modules.
prefix/include/pythonversion,	Recommended locations of the directories containing the include files
<pre>exec_prefix/include/</pre>	needed for developing Python extensions and embedding the inter-
pythonversion	preter.

2.4 Miscellaneous

To easily use Python scripts on Unix, you need to make them executable, e.g. with

```
$ chmod +x script
```

and put an appropriate Shebang line at the top of the script. A good choice is usually

```
#!/usr/bin/env python3
```

which searches for the Python interpreter in the whole PATH. However, some Unices may not have the **env** command, so you may need to hardcode /usr/bin/python3 as the interpreter path.

To use shell commands in your Python scripts, look at the subprocess module.

2.5 Custom OpenSSL

1. To use your vendor's OpenSSL configuration and system trust store, locate the directory with openssl.cnf file or symlink in /etc. On most distribution the file is either in /etc/ssl or /etc/pki/tls. The directory should also contain a cert.pem file and/or a certs directory.

```
$ find /etc/ -name openssl.cnf -printf "%h\n"
/etc/ssl
```

2. Download, build, and install OpenSSL. Make sure you use install_sw and not install. The install_sw target does not override openssl.cnf.

3. Build Python with custom OpenSSL (see the configure --with-openssl and --with-openssl-rpath options)

```
$ pushd python-3.x.x
$ ./configure -C \
    --with-openssl=/usr/local/custom-openssl \
    --with-openssl-rpath=auto \
    --prefix=/usr/local/python-3.x.x
$ make -j8
$ make altinstall
```

1 Note

Patch releases of OpenSSL have a backwards compatible ABI. You don't need to recompile Python to update OpenSSL. It's sufficient to replace the custom OpenSSL installation with a newer version.

CHAPTER

THREE

CONFIGURE PYTHON

3.1 Build Requirements

Features and minimum versions required to build CPython:

- A C11 compiler. Optional C11 features are not required.
- On Windows, Microsoft Visual Studio 2017 or later is required.
- Support for IEEE 754 floating-point numbers and floating-point Not-a-Number (NaN).
- · Support for threads.
- OpenSSL 1.1.1 is the minimum version and OpenSSL 3.0.9 is the recommended minimum version for the ssl and hashlib extension modules.
- SQLite 3.15.2 for the sqlite3 extension module.
- Tcl/Tk 8.5.12 for the tkinter module.
- Autoconf 2.71 and aclocal 1.16.5 are required to regenerate the configure script.

Changed in version 3.1: Tcl/Tk version 8.3.1 is now required.

Changed in version 3.5: On Windows, Visual Studio 2015 or later is now required. Tcl/Tk version 8.4 is now required.

Changed in version 3.6: Selected C99 features are now required, like <stdint.h> and static inline functions.

Changed in version 3.7: Thread support and OpenSSL 1.0.2 are now required.

Changed in version 3.10: OpenSSL 1.1.1 is now required. Require SQLite 3.7.15.

Changed in version 3.11: C11 compiler, IEEE 754 and NaN support are now required. On Windows, Visual Studio 2017 or later is required. Tcl/Tk version 8.5.12 is now required for the tkinter module.

Changed in version 3.13: Autoconf 2.71, aclocal 1.16.5 and SQLite 3.15.2 are now required.

See also PEP 7 "Style Guide for C Code" and PEP 11 "CPython platform support".

3.2 Generated files

To reduce build dependencies, Python source code contains multiple generated files. Commands to regenerate all generated files:

```
make regen-all
make regen-stdlib-module-names
make regen-limited-abi
make regen-configure
```

The Makefile.pre.in file documents generated files, their inputs, and tools used to regenerate them. Search for regen-* make targets.

3.2.1 configure script

The make regen-configure command regenerates the aclocal.m4 file and the configure script using the Tools/build/regen-configure.sh shell script which uses an Ubuntu container to get the same tools versions and have a reproducible output.

The container is optional, the following command can be run locally:

```
autoreconf -ivf -Werror
```

The generated files can change depending on the exact autoconf-archive, aclocal and pkg-config versions.

3.3 Configure Options

List all configure script options using:

```
./configure --help
```

See also the Misc/SpecialBuilds.txt in the Python source distribution.

3.3.1 General Options

--enable-loadable-sqlite-extensions

Support loadable extensions in the _sqlite extension module (default is no) of the sqlite3 module.

See the sqlite3.Connection.enable_load_extension() method of the sqlite3 module.

Added in version 3.6.

--disable-ipv6

Disable IPv6 support (enabled by default if supported), see the socket module.

--enable-big-digits=[15|30]

Define the size in bits of Python int digits: 15 or 30 bits.

By default, the digit size is 30.

Define the ${\tt PYLONG_BITS_IN_DIGIT}$ to 15 or 30.

See sys.int_info.bits_per_digit.

--with-suffix=SUFFIX

Set the Python executable suffix to SUFFIX.

The default suffix is .exe on Windows and macOS (python.exe executable), .js on Emscripten node, . html on Emscripten browser, .wasm on WASI, and an empty string on other platforms (python executable).

Changed in version 3.11: The default suffix on WASM platform is one of .js, .html or .wasm.

--with-tzpath=<list of absolute paths separated by pathsep>

Select the default time zone search path for zoneinfo.TZPATH. See the Compile-time configuration of the zoneinfo module.

Default: /usr/share/zoneinfo:/usr/lib/zoneinfo:/usr/share/lib/zoneinfo:/etc/zoneinfo.

See os.pathsep path separator.

Added in version 3.9.

--without-decimal-contextvar

Build the _decimal extension module using a thread-local context rather than a coroutine-local context (default), see the decimal module.

See decimal. HAVE_CONTEXTVAR and the contextvars module.

Added in version 3.9.

--with-dbmliborder=<list of backend names>

Override order to check db backends for the dbm module

A valid value is a colon (:) separated string with the backend names:

- ndbm;
- gdbm;
- bdb.

--without-c-locale-coercion

Disable C locale coercion to a UTF-8 based locale (enabled by default).

Don't define the PY_COERCE_C_LOCALE macro.

See PYTHONCOERCECLOCALE and the PEP 538.

--without-freelists

Disable all freelists except the empty tuple singleton.

Added in version 3.11.

--with-platlibdir=DIRNAME

Python library directory name (default is lib).

Fedora and SuSE use lib64 on 64-bit platforms.

See sys.platlibdir.

Added in version 3.9.

--with-wheel-pkg-dir=PATH

Directory of wheel packages used by the ensurepip module (none by default).

Some Linux distribution packaging policies recommend against bundling dependencies. For example, Fedora installs wheel packages in the /usr/share/python-wheels/ directory and don't install the ensurepip. _bundled package.

Added in version 3.10.

--with-pkg-config=[check|yes|no]

Whether configure should use **pkg-config** to detect build dependencies.

- check (default): pkg-config is optional
- yes: **pkg-config** is mandatory
- no: configure does not use pkg-config even when present

Added in version 3.11.

--enable-pystats

Turn on internal Python performance statistics gathering.

By default, statistics gathering is off. Use python3 -X pystats command or set PYTHONSTATS=1 environment variable to turn on statistics gathering at Python startup.

At Python exit, dump statistics if statistics gathering was on and not cleared.

Effects:

- Add -X pystats command line option.
- Add PYTHONSTATS environment variable.
- Define the Py_STATS macro.
- Add functions to the sys module:

- sys._stats_on(): Turns on statistics gathering.
- sys._stats_off(): Turns off statistics gathering.
- sys._stats_clear(): Clears the statistics.
- sys._stats_dump(): Dump statistics to file, and clears the statistics.

The statistics will be dumped to a arbitrary (probably unique) file in /tmp/py_stats/ (Unix) or C:\temp\py_stats\ (Windows). If that directory does not exist, results will be printed on stderr.

Use Tools/scripts/summarize_stats.py to read the stats.

Statistics:

- Opcode:
 - Specialization: success, failure, hit, deferred, miss, deopt, failures;
 - Execution count;
 - Pair count.
- Call:
 - Inlined Python calls;
 - PyEval calls;
 - Frames pushed;
 - Frame object created;
 - Eval calls: vector, generator, legacy, function VECTORCALL, build class, slot, function "ex", API, method.
- Object:
 - incref and decref;
 - interpreter incref and decref;
 - allocations: all, 512 bytes, 4 kiB, big;
 - free;
 - to/from free lists;
 - dictionary materialized/dematerialized;
 - type cache;
 - optimization attempts;
 - optimization traces created/executed;
 - uops executed.
- · Garbage collector:
 - Garbage collections;
 - Objects visited;
 - Objects collected.

Added in version 3.11.

--disable-gil

Enables **experimental** support for running Python without the *global interpreter lock* (GIL): free threading build.

Defines the Py_GIL_DISABLED macro and adds "t" to sys.abiflags.

See whatsnew313-free-threaded-cpython for more detail.

Added in version 3.13.

--enable-experimental-jit=[no|yes|yes-off|interpreter]

Indicate how to integrate the JIT compiler.

- no build the interpreter without the JIT.
- yes build the interpreter with the JIT.
- yes-off build the interpreter with the JIT but disable it by default.
- interpreter build the interpreter without the JIT, but with the tier 2 enabled interpreter.

 $By\ convention, \ -\text{-enable-experimental-jit}\ is\ a\ shorthand\ for\ -\text{-enable-experimental-jit=yes}.$

Added in version 3.13.

PKG CONFIG

Path to pkg-config utility.

PKG_CONFIG_LIBDIR

PKG_CONFIG_PATH

pkg-config options.

3.3.2 C compiler options

CC

C compiler command.

CFLAGS

C compiler flags.

CPP

C preprocessor command.

CPPFLAGS

C preprocessor flags, e.g. -Iinclude_dir.

3.3.3 Linker options

LDFLAGS

Linker flags, e.g. -Llibrary_directory.

LIBS

Libraries to pass to the linker, e.g. -llibrary.

MACHDEP

Name for machine-dependent library files.

3.3.4 Options for third-party dependencies

Added in version 3.11.

BZIP2_CFLAGS

BZIP2_LIBS

C compiler and linker flags to link Python to libbz2, used by bz2 module, overriding pkg-config.

CURSES_CFLAGS

CURSES_LIBS

C compiler and linker flags for libncurses or libncursesw, used by curses module, overriding pkg-config.

GDBM_CFLAGS

GDBM_LIBS

C compiler and linker flags for gdbm.

LIBB2_CFLAGS

LIBB2 LIBS

C compiler and linker flags for libb2 (BLAKE2), used by hashlib module, overriding pkg-config.

LIBEDIT_CFLAGS

LIBEDIT LIBS

C compiler and linker flags for libedit, used by readline module, overriding pkg-config.

LIBFFI_CFLAGS

LIBFFI_LIBS

C compiler and linker flags for libffi, used by ctypes module, overriding pkg-config.

LIBMPDEC_CFLAGS

LIBMPDEC_LIBS

C compiler and linker flags for libmpdec, used by decimal module, overriding pkg-config.

On the state of the state of

These environment variables have no effect unless --with-system-libmpdec is specified.

LIBLZMA_CFLAGS

LIBLZMA_LIBS

C compiler and linker flags for liblzma, used by lzma module, overriding pkg-config.

LIBREADLINE_CFLAGS

LIBREADLINE LIBS

C compiler and linker flags for libreadline, used by readline module, overriding pkg-config.

LIBSQLITE3_CFLAGS

LIBSQLITE3_LIBS

C compiler and linker flags for libsqlite3, used by sqlite3 module, overriding pkg-config.

LIBUUID CFLAGS

LIBUUID_LIBS

C compiler and linker flags for libuuid, used by uuid module, overriding pkg-config.

PANEL_CFLAGS

PANEL_LIBS

C compiler and linker flags for PANEL, overriding pkg-config.

C compiler and linker flags for libpanel or libpanelw, used by curses.panel module, overriding pkg-config.

TCLTK_CFLAGS

TCLTK_LIBS

C compiler and linker flags for TCLTK, overriding pkg-config.

ZLIB_CFLAGS

ZLIB_LIBS

C compiler and linker flags for libzlib, used by gzip module, overriding pkg-config.

3.3.5 WebAssembly Options

--with-emscripten-target=[browser|node]

Set build flavor for wasm32-emscripten.

- browser (default): preload minimal stdlib, default MEMFS.
- node: NODERAWFS and pthread support.

Added in version 3.11.

--enable-wasm-dynamic-linking

Turn on dynamic linking support for WASM.

Dynamic linking enables <code>dlopen</code>. File size of the executable increases due to limited dead code elimination and additional features.

Added in version 3.11.

--enable-wasm-pthreads

Turn on pthreads support for WASM.

Added in version 3.11.

3.3.6 Install Options

--prefix=PREFIX

Install architecture-independent files in PREFIX. On Unix, it defaults to /usr/local.

This value can be retrieved at runtime using sys.prefix.

As an example, one can use --prefix="\$HOME/.local/" to install a Python in its home directory.

--exec-prefix=EPREFIX

Install architecture-dependent files in EPREFIX, defaults to --prefix.

This value can be retrieved at runtime using sys.exec_prefix.

--disable-test-modules

Don't build nor install test modules, like the test package or the _testcapi extension module (built and installed by default).

Added in version 3.10.

--with-ensurepip=[upgrade|install|no]

Select the ${\tt ensurepip}$ command run on Python installation:

- upgrade (default): run python -m ensurepip --altinstall --upgrade command.
- \bullet install: run python -m ensurepip --altinstall command;
- no: don't run ensurepip;

Added in version 3.6.

3.3.7 Performance options

Configuring Python using --enable-optimizations --with-lto (PGO + LTO) is recommended for best performance. The experimental --enable-bolt flag can also be used to improve performance.

--enable-optimizations

Enable Profile Guided Optimization (PGO) using PROFILE_TASK (disabled by default).

The C compiler Clang requires <code>llvm-profdata</code> program for PGO. On macOS, GCC also requires it: GCC is just an alias to Clang on macOS.

Disable also semantic interposition in libpython if --enable-shared and GCC is used: add -fno-semantic-interposition to the compiler and linker flags.



During the build, you may encounter compiler warnings about profile data not being available for some source files. These warnings are harmless, as only a subset of the code is exercised during profile data acquisition. To disable these warnings on Clang, manually suppress them by adding -Wno-profile-instr-unprofiled to CFLAGS.

Added in version 3.6.

Changed in version 3.10: Use -fno-semantic-interposition on GCC.

PROFILE_TASK

Environment variable used in the Makefile: Python command line arguments for the PGO generation task.

Default: -m test --pgo --timeout=\$(TESTTIMEOUT).

Added in version 3.8.

Changed in version 3.13: Task failure is no longer ignored silently.

--with-lto=[full|thin|no|yes]

Enable Link Time Optimization (LTO) in any build (disabled by default).

The C compiler Clang requires <code>llvm-ar</code> for LTO (ar on macOS), as well as an LTO-aware linker (ld.gold or lld).

Added in version 3.6.

Added in version 3.11: To use ThinLTO feature, use --with-lto=thin on Clang.

Changed in version 3.12: Use ThinLTO as the default optimization policy on Clang if the compiler accepts the flag.

--enable-bolt

Enable usage of the BOLT post-link binary optimizer (disabled by default).

BOLT is part of the LLVM project but is not always included in their binary distributions. This flag requires that <code>llvm-bolt</code> and <code>merge-fdata</code> are available.

BOLT is still a fairly new project so this flag should be considered experimental for now. Because this tool operates on machine code its success is dependent on a combination of the build environment + the other optimization configure args + the CPU architecture, and not all combinations are supported. BOLT versions before LLVM 16 are known to crash BOLT under some scenarios. Use of LLVM 16 or newer for BOLT optimization is strongly encouraged.

The BOLT_INSTRUMENT_FLAGS and BOLT_APPLY_FLAGS configure variables can be defined to override the default set of arguments for llvm-bolt to instrument and apply BOLT data to binaries, respectively.

Added in version 3.12.

BOLT_APPLY_FLAGS

Arguments to 11vm-bolt when creating a BOLT optimized binary.

Added in version 3.12.

BOLT_INSTRUMENT_FLAGS

Arguments to llvm-bolt when instrumenting binaries.

Added in version 3.12.

--with-computed-gotos

Enable computed gotos in evaluation loop (enabled by default on supported compilers).

--without-mimalloc

Disable the fast mimalloc allocator (enabled by default).

See also PYTHONMALLOC environment variable.

--without-pymalloc

Disable the specialized Python memory allocator pymalloc (enabled by default).

See also PYTHONMALLOC environment variable.

--without-doc-strings

Disable static documentation strings to reduce the memory footprint (enabled by default). Documentation strings defined in Python are not affected.

Don't define the WITH_DOC_STRINGS macro.

See the PyDoc_STRVAR() macro.

--enable-profiling

Enable C-level code profiling with gprof (disabled by default).

--with-strict-overflow

Add -fstrict-overflow to the C compiler flags (by default we add -fno-strict-overflow instead).

3.3.8 Python Debug Build

A debug build is Python built with the --with-pydebug configure option.

Effects of a debug build:

- Display all warnings by default: the list of default warning filters is empty in the warnings module.
- Add d to sys.abiflags.
- Add sys.gettotalrefcount() function.
- Add -X showrefcount command line option.
- Add -d command line option and PYTHONDEBUG environment variable to debug the parser.
- Add support for the __lltrace__ variable: enable low-level tracing in the bytecode evaluation loop if the
 variable is defined.
- Install debug hooks on memory allocators to detect buffer overflow and other memory errors.
- Define Py_DEBUG and Py_REF_DEBUG macros.
- Add runtime checks: code surrounded by #ifdef Py_DEBUG and #endif. Enable assert(...) and _PyObject_ASSERT(...) assertions: don't set the NDEBUG macro (see also the --with-assertions configure option). Main runtime checks:
 - Add sanity checks on the function arguments.
 - Unicode and int objects are created with their memory filled with a pattern to detect usage of uninitialized objects.
 - Ensure that functions which can clear or replace the current exception are not called with an exception raised.
 - Check that deallocator functions don't change the current exception.
 - The garbage collector (gc.collect() function) runs some basic checks on objects consistency.

- The Py_SAFE_DOWNCAST() macro checks for integer underflow and overflow when downcasting from wide types to narrow types.

See also the Python Development Mode and the --with-trace-refs configure option.

Changed in version 3.8: Release builds and debug builds are now ABI compatible: defining the Py_DEBUG macro no longer implies the Py_TRACE_REFS macro (see the --with-trace-refs option).

3.3.9 Debug options

--with-pydebug

Build Python in debug mode: define the Py_DEBUG macro (disabled by default).

--with-trace-refs

Enable tracing references for debugging purpose (disabled by default).

Effects:

- Define the Py_TRACE_REFS macro.
- Add sys.getobjects() function.
- Add PYTHONDUMPREFS environment variable.

The PYTHONDUMPREFS environment variable can be used to dump objects and reference counts still alive at Python exit.

Statically allocated objects are not traced.

Added in version 3.8.

Changed in version 3.13: This build is now ABI compatible with release build and debug build.

--with-assertions

Build with C assertions enabled (default is no): assert (...); and _PyObject_ASSERT(...);.

If set, the NDEBUG macro is not defined in the *OPT* compiler variable.

See also the --with-pydebug option (debug build) which also enables assertions.

Added in version 3.6.

--with-valgrind

Enable Valgrind support (default is no).

--with-dtrace

Enable DTrace support (default is no).

See Instrumenting CPython with DTrace and SystemTap.

Added in version 3.6.

--with-address-sanitizer

Enable AddressSanitizer memory error detector, asan (default is no).

Added in version 3.6.

--with-memory-sanitizer

Enable MemorySanitizer allocation error detector, msan (default is no).

Added in version 3.6.

--with-undefined-behavior-sanitizer

Enable UndefinedBehaviorSanitizer undefined behaviour detector, ubsan (default is no).

Added in version 3.6.

--with-thread-sanitizer

Enable ThreadSanitizer data race detector, tsan (default is no).

Added in version 3.13.

3.3.10 Linker options

--enable-shared

Enable building a shared Python library: libpython (default is no).

--without-static-libpython

Do not build libpythonMAJOR.MINOR.a and do not install python.o (built and enabled by default).

Added in version 3.10.

3.3.11 Libraries options

--with-libs='lib1 ...'

Link against additional libraries (default is no).

--with-system-expat

Build the pyexpat module using an installed expat library (default is no).

--with-system-libmpdec

Build the _decimal extension module using an installed mpdecimal library, see the decimal module (default is yes).

Added in version 3.3.

Changed in version 3.13: Default to using the installed mpdecimal library.

Deprecated since version 3.13, will be removed in version 3.15: A copy of the mpdecimal library sources will no longer be distributed with Python 3.15.

See also

LIBMPDEC_CFLAGS and LIBMPDEC_LIBS.

--with-readline=readline|editline

Designate a backend library for the readline module.

- readline: Use readline as the backend.
- editline: Use editline as the backend.

Added in version 3.10.

--without-readline

Don't build the readline module (built by default).

Don't define the HAVE_LIBREADLINE macro.

Added in version 3.10.

--with-libm=STRING

Override libm math library to STRING (default is system-dependent).

--with-libc=STRING

Override libc C library to STRING (default is system-dependent).

--with-openssl=DIR

Root of the OpenSSL directory.

Added in version 3.7.

--with-openssl-rpath=[no|auto|DIR]

Set runtime library directory (rpath) for OpenSSL libraries:

- no (default): don't set rpath;
- auto: auto-detect rpath from --with-openss1 and pkg-config;
- DIR: set an explicit rpath.

Added in version 3.10.

3.3.12 Security Options

--with-hash-algorithm=[fnv|siphash13|siphash24]

Select hash algorithm for use in Python/pyhash.c:

- siphash13 (default);
- siphash24;
- fnv.

Added in version 3.4.

Added in version 3.11: siphash13 is added and it is the new default.

--with-builtin-hashlib-hashes=md5, sha1, sha256, sha512, sha3, blake2

Built-in hash modules:

- md5;
- sha1;
- sha256;
- sha512;
- sha3 (with shake);
- blake2.

Added in version 3.9.

--with-ssl-default-suites=[python|openssl|STRING]

Override the OpenSSL default cipher suites string:

- python (default): use Python's preferred selection;
- openssl: leave OpenSSL's defaults untouched;
- STRING: use a custom string

See the ssl module.

Added in version 3.7.

Changed in version 3.10: The settings python and STRING also set TLS 1.2 as minimum protocol version.

3.3.13 macOS Options

See Mac/README.rst.

--enable-universalsdk

--enable-universalsdk=SDKDIR

Create a universal binary build. *SDKDIR* specifies which macOS SDK should be used to perform the build (default is no).

--enable-framework

--enable-framework=INSTALLDIR

Create a Python.framework rather than a traditional Unix install. Optional *INSTALLDIR* specifies the installation path (default is no).

--with-universal-archs=ARCH

Specify the kind of universal binary that should be created. This option is only valid when --enable-universalsdk is set.

Options:

- universal2;
- 32-bit;
- 64-bit;
- 3-way;
- intel;
- intel-32;
- intel-64:
- all.

--with-framework-name=FRAMEWORK

Specify the name for the python framework on macOS only valid when --enable-framework is set (default: Python).

--with-app-store-compliance

--with-app-store-compliance=PATCH-FILE

The Python standard library contains strings that are known to trigger automated inspection tool errors when submitted for distribution by the macOS and iOS App Stores. If enabled, this option will apply the list of patches that are known to correct app store compliance. A custom patch file can also be specified. This option is disabled by default.

Added in version 3.13.

3.3.14 iOS Options

See iOS/README.rst.

--enable-framework=INSTALLDIR

Create a Python.framework. Unlike macOS, the *INSTALLDIR* argument specifying the installation path is mandatory.

--with-framework-name = FRAMEWORK

Specify the name for the framework (default: Python).

3.3.15 Cross Compiling Options

Cross compiling, also known as cross building, can be used to build Python for another CPU architecture or platform. Cross compiling requires a Python interpreter for the build platform. The version of the build Python must match the version of the cross compiled host Python.

--build=BUILD

configure for building on BUILD, usually guessed by config.guess.

```
--host=HOST
```

cross-compile to build programs to run on HOST (target platform)

--with-build-python=path/to/python

path to build python binary for cross compiling

Added in version 3.11.

CONFIG SITE=file

An environment variable that points to a file with configure overrides.

Example *config.site* file:

```
# config.site-aarch64
ac_cv_buggy_getaddrinfo=no
ac_cv_file__dev_ptmx=yes
ac_cv_file__dev_ptc=no
```

HOSTRUNNER

Program to run CPython for the host platform for cross-compilation.

Added in version 3.11.

Cross compiling example:

```
CONFIG_SITE=config.site-aarch64 ../configure \
--build=x86_64-pc-linux-gnu \
--host=aarch64-unknown-linux-gnu \
--with-build-python=../x86_64/python
```

3.4 Python Build System

3.4.1 Main files of the build system

- configure.ac => configure;
- Makefile.pre.in => Makefile (created by configure);
- pyconfig.h (created by configure);
- Modules/Setup: C extensions built by the Makefile using Module/makesetup shell script;

3.4.2 Main build steps

- C files (.c) are built as object files (.o).
- A static libpython library (.a) is created from objects files.
- $\bullet\,$ python.o and the static libpython library are linked into the final python program.
- C extensions are built by the Makefile (see Modules/Setup).

3.4.3 Main Makefile targets

make

For the most part, when rebuilding after editing some code or refreshing your checkout from upstream, all you need to do is execute make, which (per Make's semantics) builds the default target, the first one defined in the Makefile. By tradition (including in the CPython project) this is usually the all target. The configure script expands an autoconf variable, @DEF_MAKE_ALL_RULE@ to describe precisely which targets make all will build. The three choices are:

• profile-opt (configured with --enable-optimizations)

- build_wasm (configured with --with-emscripten-target)
- build_all (configured without explicitly using either of the others)

Depending on the most recent source file changes, Make will rebuild any targets (object files and executables) deemed out-of-date, including running configure again if necessary. Source/target dependencies are many and maintained manually however, so Make sometimes doesn't have all the information necessary to correctly detect all targets which need to be rebuilt. Depending on which targets aren't rebuilt, you might experience a number of problems. If you have build or test problems which you can't otherwise explain, make clean && make should work around most dependency problems, at the expense of longer build times.

make platform

Build the python program, but don't build the standard library extension modules. This generates a file named platform which contains a single line describing the details of the build platform, e.g., macosx-14.3-arm64-3.12 or linux-x86_64-3.13.

make profile-opt

Build Python using profile-guided optimization (PGO). You can use the configure ——enable—optimizations option to make this the default target of the make command (make all or just make).

make clean

Remove built files.

make distclean

In addition to the work done by make clean, remove files created by the configure script. configure will have to be run before building again.

make install

Build the all target and install Python.

make test

Build the all target and run the Python test suite with the --fast-ci option. Variables:

- TESTOPTS: additional regrtest command-line options.
- TESTPYTHONOPTS: additional Python command-line options.
- TESTTIMEOUT: timeout in seconds (default: 10 minutes).

make buildbottest

This is similar to make test, but uses the --slow-ci option and default timeout of 20 minutes, instead of --fast-ci option.

make regen-all

Regenerate (almost) all generated files. These include (but are not limited to) bytecode cases, and parser generator file. make regen-stdlib-module-names and autoconf must be run separately for the remaining *generated files*.

¹ git clean -fdx is an even more extreme way to "clean" your checkout. It removes all files not known to Git. When bug hunting using git bisect, this is recommended between probes to guarantee a completely clean build. **Use with care**, as it will delete all files not checked into Git, including your new, uncommitted work.

3.4.4 C extensions

Some C extensions are built as built-in modules, like the sys module. They are built with the Py_BUILD_CORE_BUILTIN macro defined. Built-in modules have no __file__ attribute:

```
>>> import sys
>>> sys
<module 'sys' (built-in)>
>>> sys.__file__
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
AttributeError: module 'sys' has no attribute '__file__'
```

Other C extensions are built as dynamic libraries, like the _asyncio module. They are built with the Py_BUILD_CORE_MODULE macro defined. Example on Linux x86-64:

Modules/Setup is used to generate Makefile targets to build C extensions. At the beginning of the files, C extensions are built as built-in modules. Extensions defined after the *shared* marker are built as dynamic libraries.

The PyAPI_FUNC(), PyAPI_DATA() and PyMODINIT_FUNC macros of Include/exports.h are defined differently depending if the Py_BUILD_CORE_MODULE macro is defined:

- Use Py_EXPORTED_SYMBOL if the Py_BUILD_CORE_MODULE is defined
- Use Py_IMPORTED_SYMBOL otherwise.

If the $Py_BUILD_CORE_BUILTIN$ macro is used by mistake on a C extension built as a shared library, its $PyInit_xxx$ () function is not exported, causing an ImportError on import.

3.5 Compiler and linker flags

Options set by the ./configure script and environment variables and used by Makefile.

3.5.1 Preprocessor flags

CONFIGURE CPPFLAGS

Value of CPPFLAGS variable passed to the ./configure script.

Added in version 3.6.

CPPFLAGS

(Objective) C/C++ preprocessor flags, e.g. -Iinclude_dir if you have headers in a nonstandard directory include_dir.

Both CPPFLAGS and LDFLAGS need to contain the shell's value to be able to build extension modules using the directories specified in the environment variables.

BASECPPFLAGS

Added in version 3.4.

PY_CPPFLAGS

Extra preprocessor flags added for building the interpreter object files.

```
Default: $(BASECPPFLAGS) -I. -I$(srcdir)/Include $(CONFIGURE_CPPFLAGS)
$(CPPFLAGS).
```

Added in version 3.2.

3.5.2 Compiler flags

CC

C compiler command.

Example: gcc -pthread.

CXX

C++ compiler command.

Example: g++ -pthread.

CFLAGS

C compiler flags.

CFLAGS_NODIST

CFLAGS_NODIST is used for building the interpreter and stdlib C extensions. Use it when a compiler flag should *not* be part of CFLAGS once Python is installed (gh-65320).

In particular, CFLAGS should not contain:

- the compiler flag -I (for setting the search path for include files). The -I flags are processed from left to right, and any flags in CFLAGS would take precedence over user- and package-supplied -I flags.
- hardening flags such as -Werror because distributions cannot control whether packages installed by users conform to such heightened standards.

Added in version 3.5.

COMPILEALL_OPTS

 $Options\ passed\ to\ the\ \texttt{compileall}\ command\ line\ when\ building\ PYC\ files\ in\ \texttt{make}\ \ \texttt{install}.\ Default:\ -\texttt{j0}.$

Added in version 3.12.

EXTRA_CFLAGS

Extra C compiler flags.

CONFIGURE_CFLAGS

Value of CFLAGS variable passed to the ./configure script.

Added in version 3.2.

CONFIGURE_CFLAGS_NODIST

Value of CFLAGS_NODIST variable passed to the ./configure script.

Added in version 3.5.

BASECFLAGS

Base compiler flags.

OPT

Optimization flags.

CFLAGS_ALIASING

Strict or non-strict aliasing flags used to compile Python/dtoa.c.

Added in version 3.7.

CCSHARED

Compiler flags used to build a shared library.

For example, -fPIC is used on Linux and on BSD.

CFLAGSFORSHARED

Extra C flags added for building the interpreter object files.

Default: \$ (CCSHARED) when --enable-shared is used, or an empty string otherwise.

PY CFLAGS

Default: \$(BASECFLAGS) \$(OPT) \$(CONFIGURE_CFLAGS) \$(CFLAGS) \$(EXTRA_CFLAGS).

PY_CFLAGS_NODIST

Default: \$(CONFIGURE_CFLAGS_NODIST) \$(CFLAGS_NODIST) -I\$(srcdir)/Include/internal.

Added in version 3.5.

PY_STDMODULE_CFLAGS

C flags used for building the interpreter object files.

Default: \$(PY_CFLAGS) \$(PY_CFLAGS_NODIST) \$(PY_CPPFLAGS) \$(CFLAGSFORSHARED).

Added in version 3.7.

PY_CORE_CFLAGS

Default: \$ (PY_STDMODULE_CFLAGS) -DPY_BUILD_CORE.

Added in version 3.2.

PY_BUILTIN_MODULE_CFLAGS

Compiler flags to build a standard library extension module as a built-in module, like the posix module.

Default: \$(PY_STDMODULE_CFLAGS) -DPY_BUILD_CORE_BUILTIN.

Added in version 3.8.

PURIFY

Purify command. Purify is a memory debugger program.

Default: empty string (not used).

3.5.3 Linker flags

LINKCC

Linker command used to build programs like python and _testembed.

Default: \$(PURIFY) \$(CC).

CONFIGURE_LDFLAGS

Value of LDFLAGS variable passed to the ./configure script.

Avoid assigning CFLAGS, LDFLAGS, etc. so users can use them on the command line to append to these values without stomping the pre-set values.

Added in version 3.2.

LDFLAGS_NODIST

LDFLAGS_NODIST is used in the same manner as CFLAGS_NODIST. Use it when a linker flag should *not* be part of LDFLAGS once Python is installed (gh-65320).

In particular, LDFLAGS should not contain:

• the compiler flag -L (for setting the search path for libraries). The -L flags are processed from left to right, and any flags in LDFLAGS would take precedence over user- and package-supplied -L flags.

CONFIGURE LDFLAGS NODIST

Value of LDFLAGS_NODIST variable passed to the ./configure script.

Added in version 3.8.

LDFLAGS

Linker flags, e.g. -Llib_dir if you have libraries in a nonstandard directory lib_dir.

Both CPPFLAGS and LDFLAGS need to contain the shell's value to be able to build extension modules using the directories specified in the environment variables.

LIBS

Linker flags to pass libraries to the linker when linking the Python executable.

Example: -lrt.

LDSHARED

Command to build a shared library.

Default: @LDSHARED@ \$ (PY_LDFLAGS).

BLDSHARED

Command to build libpython shared library.

Default: @BLDSHARED@ \$(PY_CORE_LDFLAGS).

PY_LDFLAGS

 $Default: \verb§(CONFIGURE_LDFLAGS) \verb§(LDFLAGS).$

PY_LDFLAGS_NODIST

Default: \$(CONFIGURE_LDFLAGS_NODIST) \$(LDFLAGS_NODIST).

Added in version 3.8.

PY_CORE_LDFLAGS

Linker flags used for building the interpreter object files.

Added in version 3.8.

USING PYTHON ON WINDOWS

This document aims to give an overview of Windows-specific behaviour you should know about when using Python on Microsoft Windows.

Unlike most Unix systems and services, Windows does not include a system supported installation of Python. To make Python available, the CPython team has compiled Windows installers with every release for many years. These installers are primarily intended to add a per-user installation of Python, with the core interpreter and library being used by a single user. The installer is also able to install for all users of a single machine, and a separate ZIP file is available for application-local distributions.

As specified in **PEP 11**, a Python release only supports a Windows platform while Microsoft considers the platform under extended support. This means that Python 3.13 supports Windows 8.1 and newer. If you require Windows 7 support, please install Python 3.8.

There are a number of different installers available for Windows, each with certain benefits and downsides.

The full installer contains all components and is the best option for developers using Python for any kind of project.

The Microsoft Store package is a simple installation of Python that is suitable for running scripts and packages, and using IDLE or other development environments. It requires Windows 10 and above, but can be safely installed without corrupting other programs. It also provides many convenient commands for launching Python and its tools.

The nuget.org packages are lightweight installations intended for continuous integration systems. It can be used to build Python packages or run scripts, but is not updateable and has no user interface tools.

The embeddable package is a minimal package of Python suitable for embedding into a larger application.

4.1 The full installer

4.1.1 Installation steps

Four Python 3.13 installers are available for download - two each for the 32-bit and 64-bit versions of the interpreter. The *web installer* is a small initial download, and it will automatically download the required components as necessary. The *offline installer* includes the components necessary for a default installation and only requires an internet connection for optional features. See *Installing Without Downloading* for other ways to avoid downloading during installation.

After starting the installer, one of two options may be selected:



If you select "Install Now":

- You will *not* need to be an administrator (unless a system update for the C Runtime Library is required or you install the *Python Launcher for Windows* for all users)
- Python will be installed into your user directory
- The Python Launcher for Windows will be installed according to the option at the bottom of the first page
- The standard library, test suite, launcher and pip will be installed
- If selected, the install directory will be added to your PATH
- · Shortcuts will only be visible for the current user

Selecting "Customize installation" will allow you to select the features to install, the installation location and other options or post-install actions. To install debugging symbols or binaries, you will need to use this option.

To perform an all-users installation, you should select "Customize installation". In this case:

- You may be required to provide administrative credentials or approval
- Python will be installed into the Program Files directory
- The Python Launcher for Windows will be installed into the Windows directory
- Optional features may be selected during installation
- The standard library can be pre-compiled to bytecode
- If selected, the install directory will be added to the system PATH
- Shortcuts are available for all users

4.1.2 Removing the MAX PATH Limitation

Windows historically has limited path lengths to 260 characters. This meant that paths longer than this would not resolve and errors would result.

In the latest versions of Windows, this limitation can be expanded to approximately 32,000 characters. Your administrator will need to activate the "Enable Win32 long paths" group policy, or set <code>LongPathsEnabled</code> to 1 in the registry key <code>HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem</code>.

This allows the open () function, the os module and most other path functionality to accept and return paths longer than 260 characters.

After changing the above option, no further configuration is required.

Changed in version 3.6: Support for long paths was enabled in Python.

4.1.3 Installing Without UI

All of the options available in the installer UI can also be specified from the command line, allowing scripted installers to replicate an installation on many machines without user interaction. These options may also be set without suppressing the UI in order to change some of the defaults.

The following options (found by executing the installer with /?) can be passed into the installer:

Name	Description
/passive	to display progress without requiring user interaction
/quiet	to install/uninstall without displaying any UI
/simple	to prevent user customization
/uninstall	to remove Python (without confirmation)
/layout [directory]	to pre-download all components
/log [filename]	to specify log files location

All other options are passed as name=value, where the value is usually 0 to disable a feature, 1 to enable a feature, or a path. The full list of available options is shown below.

4.1. The full installer 45

Sim-

A custom message to display when (empty)

Name	Description	Default
InstallAl- lUsers	Perform a system-wide installation.	0
TargetDir Default-	The installation directory	Selected based on InstallAllUsers
AllUser- sTarget- Dir	The default installation directory for all-user installs	%ProgramFiles%\Python X.Y or %ProgramFiles(x86)%\Python X.Y
De- faultJust- ForMeTar- getDir	The default install directory for just-for-me installs	%LocalAppData%\Programs\Python\PythonXY or %LocalAppData%\Programs\Python\PythonXY-32 or %LocalAppData%\Programs\Python\PythonXY-64
Default- Custom- Target- Dir	The default custom install directory displayed in the UI	(empty)
Associ- ateFiles	Create file associations if the launcher is also installed.	1
Com- pileAll	Compile all .py files to .pyc.	0
Prepend- Path	Prepend install and Scripts directories to PATH and add .PY to PATHEXT	0
Append- Path	Append install and Scripts directories to PATH and add .PY to PATHEXT	0
Shortcuts	Create shortcuts for the interpreter, documentation and IDLE if installed.	1
In- clude_doc	Install Python manual	1
In- clude_debu	Install debug binaries	0
In- clude_dev	Install developer headers and libraries. Omitting this may lead to an unusable installation.	1
In- clude_exe	Install python.exe and related files. Omitting this may lead to an unusable installation.	1
In- clude_launc		1
Install- Launcher- AllUsers	Installs the launcher for all users. Also requires Include_launcher to be set to 1	1
In- clude_lib	Install standard library and extension modules. Omitting this may lead to an unusable installation.	1
In- clude_pip	Install bundled pip and setuptools	1
In- clude_symł	Install debugging symbols (*. pdb)	0
In- clude_tcltk	Install Tcl/Tk support and IDLE	1
In- clude_test	Install standard library test suite	1
In- clude_tools	Install utility scripts	1
LauncherO	Only installs the launcher. This	Chapter 4. Using Python on Windows
Simple- Install	will override most other options. Disable most install UI	
nioun		

For example, to silently install a default, system-wide Python installation, you could use the following command (from an elevated command prompt):

```
python-3.9.0.exe /quiet InstallAllUsers=1 PrependPath=1 Include_test=0
```

To allow users to easily install a personal copy of Python without the test suite, you could provide a shortcut with the following command. This will display a simplified initial page and disallow customization:

```
python-3.9.0.exe InstallAllUsers=0 Include_launcher=0 Include_test=0
SimpleInstall=1 SimpleInstallDescription="Just for me, no test suite."
```

(Note that omitting the launcher also omits file associations, and is only recommended for per-user installs when there is also a system-wide installation that included the launcher.)

The options listed above can also be provided in a file named unattend.xml alongside the executable. This file specifies a list of options and values. When a value is provided as an attribute, it will be converted to a number if possible. Values provided as element text are always left as strings. This example file sets the same options as the previous example:

```
<Options>
     <Option Name="InstallAllUsers" Value="no" />
     <Option Name="Include_launcher" Value="0" />
     <Option Name="Include_test" Value="no" />
     <Option Name="SimpleInstall" Value="yes" />
     <Option Name="SimpleInstallDescription">Just for me, no test suite</Option>
</Options>
```

4.1.4 Installing Without Downloading

As some features of Python are not included in the initial installer download, selecting those features may require an internet connection. To avoid this need, all possible components may be downloaded on-demand to create a complete *layout* that will no longer require an internet connection regardless of the selected features. Note that this download may be bigger than required, but where a large number of installations are going to be performed it is very useful to have a locally cached copy.

Execute the following command from Command Prompt to download all possible required files. Remember to substitute python-3.9.0.exe for the actual name of your installer, and to create layouts in their own directories to avoid collisions between files with the same name.

```
python-3.9.0.exe /layout [optional target directory]
```

You may also specify the /quiet option to hide the progress display.

4.1.5 Modifying an install

Once Python has been installed, you can add or remove features through the Programs and Features tool that is part of Windows. Select the Python entry and choose "Uninstall/Change" to open the installer in maintenance mode.

"Modify" allows you to add or remove features by modifying the checkboxes - unchanged checkboxes will not install or remove anything. Some options cannot be changed in this mode, such as the install directory; to modify these, you will need to remove and then reinstall Python completely.

"Repair" will verify all the files that should be installed using the current settings and replace any that have been removed or modified.

"Uninstall" will remove Python entirely, with the exception of the *Python Launcher for Windows*, which has its own entry in Programs and Features.

4.1. The full installer 47

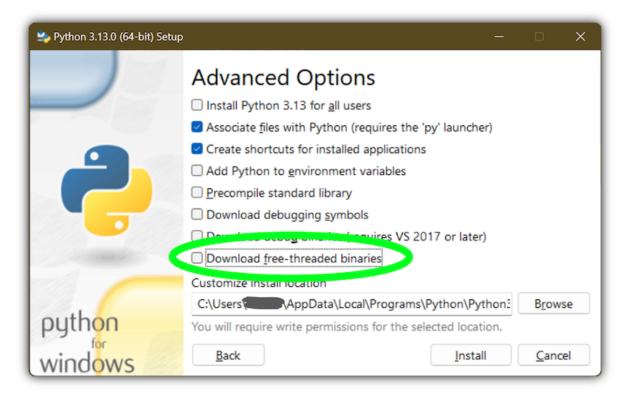
4.1.6 Installing Free-threaded Binaries

Added in version 3.13: (Experimental)



Everything described in this section is considered experimental, and should be expected to change in future releases.

To install pre-built binaries with free-threading enabled (see **PEP 703**), you should select "Customize installation". The second page of options includes the "Download free-threaded binaries" checkbox.



Selecting this option will download and install additional binaries to the same location as the main Python install. The main executable is called python3.13t.exe, and other binaries either receive a t suffix or a full ABI suffix. Python source files and bundled third-party dependencies are shared with the main install.

The free-threaded version is registered as a regular Python install with the tag 3.13t (with a -32 or -arm64 suffix as normal for those platforms). This allows tools to discover it, and for the *Python Launcher for Windows* to support py. exe -3.13t. Note that the launcher will interpret py.exe -3 (or a python3 shebang) as "the latest 3.x install", which will prefer the free-threaded binaries over the regular ones, while py.exe -3.13 will not. If you use the short style of option, you may prefer to not install the free-threaded binaries at this time.

To specify the install option at the command line, use Include_freethreaded=1. See *Installing Without Downloading* for instructions on pre-emptively downloading the additional binaries for offline install. The options to include debug symbols and binaries also apply to the free-threaded builds.

Free-threaded binaries are also available on nuget.org.

4.2 The Microsoft Store package

Added in version 3.7.2.

The Microsoft Store package is an easily installable Python interpreter that is intended mainly for interactive use, for example, by students.

To install the package, ensure you have the latest Windows 10 updates and search the Microsoft Store app for "Python 3.13". Ensure that the app you select is published by the Python Software Foundation, and install it.

Warning

Python will always be available for free on the Microsoft Store. If you are asked to pay for it, you have not selected the correct package.

After installation, Python may be launched by finding it in Start. Alternatively, it will be available from any Command Prompt or PowerShell session by typing python. Further, pip and IDLE may be used by typing pip or idle. IDLE can also be found in Start.

All three commands are also available with version number suffixes, for example, as python3.exe and python3. x.exe as well as python.exe (where 3.x is the specific version you want to launch, such as 3.13). Open "Manage App Execution Aliases" through Start to select which version of Python is associated with each command. It is recommended to make sure that pip and idle are consistent with whichever version of python is selected.

Virtual environments can be created with python -m venv and activated and used as normal.

If you have installed another version of Python and added it to your PATH variable, it will be available as python.exe rather than the one from the Microsoft Store. To access the new installation, use python3.exe or python3.x.exe.

The py, exe launcher will detect this Python installation, but will prefer installations from the traditional installer.

To remove Python, open Settings and use Apps and Features, or else find Python in Start and right-click to select Uninstall. Uninstalling will remove all packages you installed directly into this Python installation, but will not remove any virtual environments

4.2.1 Known issues

Redirection of local data, registry, and temporary paths

Because of restrictions on Microsoft Store apps, Python scripts may not have full write access to shared locations such as TEMP and the registry. Instead, it will write to a private copy. If your scripts must modify the shared locations, you will need to install the full installer.

At runtime, Python will use a private copy of well-known Windows folders and the registry. example, if the environment variable %APPDATA% is c:\Users\<user>\AppData\, then when writing to C:\Users\<user>\AppData\Local will write to C:\Users\<user>\AppData\Local\Packages\ PythonSoftwareFoundation.Python.3.8_qbz5n2kfra8p0\LocalCache\Local\.

When reading files, Windows will return the file from the private folder, or if that does not exist, the real Windows directory. For example reading C:\Windows\System32 returns the contents of C:\Windows\System32 plus the contents of C:\Program Files\WindowsApps\package_name\VFS\SystemX86.

You can find the real path of any existing file using os.path.realpath():

```
>>> import os
>>> test_file = 'C:\\Users\\example\\AppData\\Local\\test.txt'
>>> os.path.realpath(test_file)
'C:\\Users\\example\\AppData\\Local\\Packages\\PythonSoftwareFoundation.Python.3.8_
→qbz5n2kfra8p0\\LocalCache\\Local\\test.txt'
```

When writing to the Windows Registry, the following behaviors exist:

- Reading from HKLM\\Software is allowed and results are merged with the registry.dat file in the package.
- Writing to HKLM\\Software is not allowed if the corresponding key/value exists, i.e. modifying existing
- Writing to HKLM\\Software is allowed as long as a corresponding key/value does not exist in the package and the user has the correct access permissions.

For more detail on the technical basis for these limitations, please consult Microsoft's documentation on packaged full-trust apps, currently available at docs.microsoft.com/en-us/windows/msix/desktop/desktop-to-uwp-behind-the-scenes

4.3 The nuget.org packages

Added in version 3.5.2.

The nuget.org package is a reduced size Python environment intended for use on continuous integration and build systems that do not have a system-wide install of Python. While nuget is "the package manager for .NET", it also works perfectly fine for packages containing build-time tools.

Visit nuget.org for the most up-to-date information on using nuget. What follows is a summary that is sufficient for Python developers.

The nuget.exe command line tool may be downloaded directly from https://aka.ms/nugetclidl, for example, using curl or PowerShell. With the tool, the latest version of Python for 64-bit or 32-bit machines is installed using:

```
nuget.exe install python -ExcludeVersion -OutputDirectory .
nuget.exe install pythonx86 -ExcludeVersion -OutputDirectory .
```

To select a particular version, add a -Version 3.x.y. The output directory may be changed from ., and the package will be installed into a subdirectory. By default, the subdirectory is named the same as the package, and without the -ExcludeVersion option this name will include the specific version installed. Inside the subdirectory is a tools directory that contains the Python installation:

```
# Without -ExcludeVersion
> .\python.3.5.2\tools\python.exe -V
Python 3.5.2

# With -ExcludeVersion
> .\python\tools\python.exe -V
Python 3.5.2
```

In general, nuget packages are not upgradeable, and newer versions should be installed side-by-side and referenced using the full path. Alternatively, delete the package directory manually and install it again. Many CI systems will do this automatically if they do not preserve files between builds.

Alongside the tools directory is a build\native directory. This contains a MSBuild properties file python. props that can be used in a C++ project to reference the Python install. Including the settings will automatically use the headers and import libraries in your build.

The package information pages on nuget.org are www.nuget.org/packages/python for the 64-bit version, www.nuget.org/packages/pythonx86 for the 32-bit version, and www.nuget.org/packages/pythonarm64 for the ARM64 version

4.3.1 Free-threaded packages

Added in version 3.13: (Experimental)



Everything described in this section is considered experimental, and should be expected to change in future

Packages containing free-threaded binaries are named python-freethreaded for the 64-bit version, pythonx86-freethreaded for the 32-bit version, and pythonarm64-freethreaded for the ARM64 version. These packages contain both the python3.13t.exe and python.exe entry points, both of which run free threaded.

4.4 The embeddable package

Added in version 3.5.

The embedded distribution is a ZIP file containing a minimal Python environment. It is intended for acting as part of another application, rather than being directly accessed by end-users.

When extracted, the embedded distribution is (almost) fully isolated from the user's system, including environment variables, system registry settings, and installed packages. The standard library is included as pre-compiled and optimized .pyc files in a ZIP, and python3.dll, python3.dll, python.exe and pythonw.exe are all provided. Tcl/tk (including all dependents, such as Idle), pip and the Python documentation are not included.



1 Note

The embedded distribution does not include the Microsoft C Runtime and it is the responsibility of the application installer to provide this. The runtime may have already been installed on a user's system previously or automatically via Windows Update, and can be detected by finding ucrtbase.dll in the system directory.

Third-party packages should be installed by the application installer alongside the embedded distribution. Using pip to manage dependencies as for a regular Python installation is not supported with this distribution, though with some care it may be possible to include and use pip for automatic updates. In general, third-party packages should be treated as part of the application ("vendoring") so that the developer can ensure compatibility with newer versions before providing updates to users.

The two recommended use cases for this distribution are described below.

4.4.1 Python Application

An application written in Python does not necessarily require users to be aware of that fact. The embedded distribution may be used in this case to include a private version of Python in an install package. Depending on how transparent it should be (or conversely, how professional it should appear), there are two options.

Using a specialized executable as a launcher requires some coding, but provides the most transparent experience for users. With a customized launcher, there are no obvious indications that the program is running on Python: icons can be customized, company and version information can be specified, and file associations behave properly. In most cases, a custom launcher should simply be able to call Py_Main with a hard-coded command line.

The simpler approach is to provide a batch file or generated shortcut that directly calls the python. exe or pythonw. exe with the required command-line arguments. In this case, the application will appear to be Python and not its actual name, and users may have trouble distinguishing it from other running Python processes or file associations.

With the latter approach, packages should be installed as directories alongside the Python executable to ensure they are available on the path. With the specialized launcher, packages can be located in other locations as there is an opportunity to specify the search path before launching the application.

4.4.2 Embedding Python

Applications written in native code often require some form of scripting language, and the embedded Python distribution can be used for this purpose. In general, the majority of the application is in native code, and some part will either invoke python. exe or directly use python3.dll. For either case, extracting the embedded distribution to a subdirectory of the application installation is sufficient to provide a loadable Python interpreter.

As with the application use, packages can be installed to any location as there is an opportunity to specify search paths before initializing the interpreter. Otherwise, there is no fundamental differences between using the embedded distribution and a regular installation.

4.5 Alternative bundles

Besides the standard CPython distribution, there are modified packages including additional functionality. The following is a list of popular versions and their key features:

ActivePython

Installer with multi-platform compatibility, documentation, PyWin32

Anaconda

Popular scientific modules (such as numpy, scipy and pandas) and the conda package manager.

Enthought Deployment Manager

"The Next Generation Python Environment and Package Manager".

Previously Enthought provided Canopy, but it reached end of life in 2016.

WinPython

Windows-specific distribution with prebuilt scientific packages and tools for building packages.

Note that these packages may not include the latest versions of Python or other libraries, and are not maintained or supported by the core Python team.

4.6 Configuring Python

To run Python conveniently from a command prompt, you might consider changing some default environment variables in Windows. While the installer provides an option to configure the PATH and PATHEXT variables for you, this is only reliable for a single, system-wide installation. If you regularly use multiple versions of Python, consider using the *Python Launcher for Windows*.

4.6.1 Excursus: Setting environment variables

Windows allows environment variables to be configured permanently at both the User level and the System level, or temporarily in a command prompt.

To temporarily set environment variables, open Command Prompt and use the set command:

```
C:\>set PATH=C:\Program Files\Python 3.9;%PATH%
C:\>set PYTHONPATH=%PYTHONPATH%;C:\My_python_lib
C:\>python
```

These changes will apply to any further commands executed in that console, and will be inherited by any applications started from the console.

Including the variable name within percent signs will expand to the existing value, allowing you to add your new value at either the start or the end. Modifying PATH by adding the directory containing **python.exe** to the start is a common way to ensure the correct version of Python is launched.

To permanently modify the default environment variables, click Start and search for 'edit environment variables', or open System properties, *Advanced system settings* and click the *Environment Variables* button. In this dialog, you can add or modify User and System variables. To change System variables, you need non-restricted access to your machine (i.e. Administrator rights).



Windows will concatenate User variables *after* System variables, which may cause unexpected results when modifying PATH.

The PYTHONPATH variable is used by all versions of Python, so you should not permanently configure it unless the listed paths only include code that is compatible with all of your installed Python versions.

See also

https://learn.microsoft.com/windows/win32/procthread/environment-variables
Overview of environment variables on Windows

https://learn.microsoft.com/windows-server/administration/windows-commands/set_1
The set command, for temporarily modifying environment variables

https://learn.microsoft.com/windows-server/administration/windows-commands/setx The setx command, for permanently modifying environment variables

4.6.2 Finding the Python executable

Changed in version 3.5.

Besides using the automatically created start menu entry for the Python interpreter, you might want to start Python in the command prompt. The installer has an option to set that up for you.

On the first page of the installer, an option labelled "Add Python to PATH" may be selected to have the installer add the install location into the PATH. The location of the Scripts\ folder is also added. This allows you to type python to run the interpreter, and pip for the package installer. Thus, you can also execute your scripts with command line options, see *Command line* documentation.

If you don't enable this option at install time, you can always re-run the installer, select Modify, and enable it. Alternatively, you can manually modify the PATH using the directions in *Excursus: Setting environment variables*. You need to set your PATH environment variable to include the directory of your Python installation, delimited by a semicolon from other entries. An example variable could look like this (assuming the first two entries already existed):

C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\Python 3.9

4.7 UTF-8 mode

Added in version 3.7.

Windows still uses legacy encodings for the system encoding (the ANSI Code Page). Python uses it for the default encoding of text files (e.g. locale.getencoding()).

This may cause issues because UTF-8 is widely used on the internet and most Unix systems, including WSL (Windows Subsystem for Linux).

You can use the Python UTF-8 Mode to change the default text encoding to UTF-8. You can enable the Python UTF-8 Mode via the -X utf8 command line option, or the PYTHONUTF8=1 environment variable. See PYTHONUTF8 for enabling UTF-8 mode, and Excursus: Setting environment variables for how to modify environment variables.

When the Python UTF-8 Mode is enabled, you can still use the system encoding (the ANSI Code Page) via the "mbcs" codec.

Note that adding PYTHONUTF8=1 to the default environment variables will affect all Python 3.7+ applications on your system. If you have any Python 3.7+ applications which rely on the legacy system encoding, it is recommended to set the environment variable temporarily or use the -X utf8 command line option.



Even when UTF-8 mode is disabled, Python uses UTF-8 by default on Windows for:

- Console I/O including standard I/O (see PEP 528 for details).
- The filesystem encoding (see PEP 529 for details).

4.7. UTF-8 mode 53

4.8 Python Launcher for Windows

Added in version 3.3.

The Python launcher for Windows is a utility which aids in locating and executing of different Python versions. It allows scripts (or the command-line) to indicate a preference for a specific Python version, and will locate and execute that version.

Unlike the PATH variable, the launcher will correctly select the most appropriate version of Python. It will prefer peruser installations over system-wide ones, and orders by language version rather than using the most recently installed version.

The launcher was originally specified in PEP 397.

4.8.1 Getting started

From the command-line

Changed in version 3.6.

System-wide installations of Python 3.3 and later will put the launcher on your PATH. The launcher is compatible with all available versions of Python, so it does not matter which version is installed. To check that the launcher is available, execute the following command in Command Prompt:

```
ру
```

You should find that the latest version of Python you have installed is started - it can be exited as normal, and any additional command-line arguments specified will be sent directly to Python.

If you have multiple versions of Python installed (e.g., 3.7 and 3.13) you will have noticed that Python 3.13 was started - to launch Python 3.7, try the command:

```
py -3.7
```

If you want the latest version of Python 2 you have installed, try the command:

```
py -2
```

If you see the following error, you do not have the launcher installed:

```
'py' is not recognized as an internal or external command, operable program or batch file.
```

The command:

```
py --list
```

displays the currently installed version(s) of Python.

The -x.y argument is the short form of the -V:Company/Tag argument, which allows selecting a specific Python runtime, including those that may have come from somewhere other than python.org. Any runtime registered by following **PEP 514** will be discoverable. The --list command lists all available runtimes using the -V:Company/Tag argument, which allows selecting a specific Python runtime, including those that may have come from somewhere other than python.org. Any runtime registered by following **PEP 514** will be discoverable. The --list command lists all available runtimes using the -V:Company/Tag argument, which allows selecting a specific Python runtime, including those that may have come from somewhere other than python.org.

When using the -V: argument, specifying the Company will limit selection to runtimes from that provider, while specifying only the Tag will select from all providers. Note that omitting the slash implies a tag:

```
# Select any '3.*' tagged runtime
py -V:3
# Select any 'PythonCore' released runtime
py -V:PythonCore/
```

(continues on next page)

(continued from previous page)

```
# Select PythonCore's latest Python 3 runtime
py -V:PythonCore/3
```

The short form of the argument (-3) only ever selects from core Python releases, and not other distributions. However, the longer form (-V:3) will select from any.

The Company is matched on the full string, case-insensitive. The Tag is matched on either the full string, or a prefix, provided the next character is a dot or a hyphen. This allows -V:3.1 to match 3.1-32, but not 3.10. Tags are sorted using numerical ordering (3.10 is newer than 3.1), but are compared using text (-V:3.01 does not match 3.1).

Virtual environments

Added in version 3.5.

If the launcher is run with no explicit Python version specification, and a virtual environment (created with the standard library venv module or the external virtualenv tool) active, the launcher will run the virtual environment's interpreter rather than the global one. To run the global interpreter, either deactivate the virtual environment, or explicitly specify the global Python version.

From a script

Let's create a test Python script - create a file called hello.py with the following contents

```
#! python
import sys
sys.stdout.write("hello from Python %s\n" % (sys.version,))
```

From the directory in which hello.py lives, execute the command:

```
py hello.py
```

You should notice the version number of your latest Python 2.x installation is printed. Now try changing the first line to be:

```
#! python3
```

Re-executing the command should now print the latest Python 3.x information. As with the above command-line examples, you can specify a more explicit version qualifier. Assuming you have Python 3.7 installed, try changing the first line to #! python 3.7 and you should find the 3.7 version information printed.

Note that unlike interactive use, a bare "python" will use the latest version of Python 2.x that you have installed. This is for backward compatibility and for compatibility with Unix, where the command python typically refers to Python 2.

From file associations

The launcher should have been associated with Python files (i.e. .py, .pyw, .pyc files) when it was installed. This means that when you double-click on one of these files from Windows explorer the launcher will be used, and therefore you can use the same facilities described above to have the script specify the version which should be used.

The key benefit of this is that a single launcher can support multiple Python versions at the same time depending on the contents of the first line.

4.8.2 Shebang Lines

If the first line of a script file starts with #!, it is known as a "shebang" line. Linux and other Unix like operating systems have native support for such lines and they are commonly used on such systems to indicate how a script should be executed. This launcher allows the same facilities to be used with Python scripts on Windows and the examples above demonstrate their use.

To allow shebang lines in Python scripts to be portable between Unix and Windows, this launcher supports a number of 'virtual' commands to specify which interpreter to use. The supported virtual commands are:

- /usr/bin/env
- /usr/bin/python
- /usr/local/bin/python
- python

For example, if the first line of your script starts with

#! /usr/bin/python

The default Python or an active virtual environment will be located and used. As many Python scripts written to work on Unix will already have this line, you should find these scripts can be used by the launcher without modification. If you are writing a new script on Windows which you hope will be useful on Unix, you should use one of the shebang lines starting with /usr.

Any of the above virtual commands can be suffixed with an explicit version (either just the major version, or the major and minor version). Furthermore the 32-bit version can be requested by adding "-32" after the minor version. I.e. /usr/bin/python3.7-32 will request usage of the 32-bit Python 3.7. If a virtual environment is active, the version will be ignored and the environment will be used.

Added in version 3.7: Beginning with python launcher 3.7 it is possible to request 64-bit version by the "-64" suffix. Furthermore it is possible to specify a major and architecture without minor (i.e. /usr/bin/python3-64).

Changed in version 3.11: The "-64" suffix is deprecated, and now implies "any architecture that is not provably i386/32-bit". To request a specific environment, use the new -V: TAG argument with the complete tag.

Changed in version 3.13: Virtual commands referencing python now prefer an active virtual environment rather than searching PATH. This handles cases where the shebang specifies /usr/bin/env python3 but python3.exe is not present in the active environment.

The /usr/bin/env form of shebang line has one further special property. Before looking for installed Python interpreters, this form will search the executable PATH for a Python executable matching the name provided as the first argument. This corresponds to the behaviour of the Unix env program, which performs a PATH search. If an executable matching the first argument after the env command cannot be found, but the argument starts with python, it will be handled as described for the other virtual commands. The environment variable PYLAUNCHER_NO_SEARCH_PATH may be set (to any value) to skip this search of PATH.

Shebang lines that do not match any of these patterns are looked up in the <code>[commands]</code> section of the launcher's .*INI file*. This may be used to handle certain commands in a way that makes sense for your system. The name of the command must be a single argument (no spaces in the shebang executable), and the value substituted is the full path to the executable (additional arguments specified in the .INI will be quoted as part of the filename).

[commands]

/bin/xpython=C:\Program Files\XPython\python.exe

Any commands not found in the .INI file are treated as **Windows** executable paths that are absolute or relative to the directory containing the script file. This is a convenience for Windows-only scripts, such as those generated by an installer, since the behavior is not compatible with Unix-style shells. These paths may be quoted, and may include multiple arguments, after which the path to the script and any additional arguments will be appended.

4.8.3 Arguments in shebang lines

The shebang lines can also specify additional options to be passed to the Python interpreter. For example, if you have a shebang line:

#! /usr/bin/python -v

Then Python will be started with the -v option

4.8.4 Customization

Customization via INI files

Two .ini files will be searched by the launcher - py.ini in the current user's application data directory (%LOCALAPPDATA% or \$env:LocalAppData) and py.ini in the same directory as the launcher. The same .ini files are used for both the 'console' version of the launcher (i.e. py.exe) and for the 'windows' version (i.e. pyw.exe).

Customization specified in the "application directory" will have precedence over the one next to the executable, so a user, who may not have write access to the .ini file next to the launcher, can override commands in that global .ini file.

Customizing default Python versions

In some cases, a version qualifier can be included in a command to dictate which version of Python will be used by the command. A version qualifier starts with a major version number and can optionally be followed by a period ('.') and a minor version specifier. Furthermore it is possible to specify if a 32 or 64 bit implementation shall be requested by adding "-32" or "-64".

For example, a shebang line of #!python has no version qualifier, while #!python3 has a version qualifier which specifies only a major version.

If no version qualifiers are found in a command, the environment variable PY_PYTHON can be set to specify the default version qualifier. If it is not set, the default is "3". The variable can specify any value that may be passed on the command line, such as "3", "3.7", "3.7-32" or "3.7-64". (Note that the "-64" option is only available with the launcher included with Python 3.7 or newer.)

If no minor version qualifiers are found, the environment variable PY_PYTHON{major} (where {major} is the current major version qualifier as determined above) can be set to specify the full version. If no such option is found, the launcher will enumerate the installed Python versions and use the latest minor release found for the major version, which is likely, although not guaranteed, to be the most recently installed version in that family.

On 64-bit Windows with both 32-bit and 64-bit implementations of the same (major.minor) Python version installed, the 64-bit version will always be preferred. This will be true for both 32-bit and 64-bit implementations of the launcher - a 32-bit launcher will prefer to execute a 64-bit Python installation of the specified version if available. This is so the behavior of the launcher can be predicted knowing only what versions are installed on the PC and without regard to the order in which they were installed (i.e., without knowing whether a 32 or 64-bit version of Python and corresponding launcher was installed last). As noted above, an optional "-32" or "-64" suffix can be used on a version specifier to change this behaviour.

Examples:

- If no relevant options are set, the commands python and python2 will use the latest Python 2.x version installed and the command python3 will use the latest Python 3.x installed.
- The command python3.7 will not consult any options at all as the versions are fully specified.
- If PY_PYTHON=3, the commands python and python3 will both use the latest installed Python 3 version.
- If PY_PYTHON=3.7-32, the command python will use the 32-bit implementation of 3.7 whereas the command python3 will use the latest installed Python (PY_PYTHON was not considered at all as a major version was specified.)
- If PY_PYTHON=3 and PY_PYTHON3=3.7, the commands python and python3 will both use specifically 3.7

In addition to environment variables, the same settings can be configured in the .INI file used by the launcher. The section in the INI file is called [defaults] and the key name will be the same as the environment variables without the leading PY_prefix (and note that the key names in the INI file are case insensitive.) The contents of an environment variable will override things specified in the INI file.

For example:

• Setting PY_PYTHON=3.7 is equivalent to the INI file containing:

```
[defaults]
python=3.7
```

• Setting PY_PYTHON=3 and PY_PYTHON3=3.7 is equivalent to the INI file containing:

```
[defaults]
python=3
python3=3.7
```

4.8.5 Diagnostics

If an environment variable PYLAUNCHER_DEBUG is set (to any value), the launcher will print diagnostic information to stderr (i.e. to the console). While this information manages to be simultaneously verbose *and* terse, it should allow you to see what versions of Python were located, why a particular version was chosen and the exact command-line used to execute the target Python. It is primarily intended for testing and debugging.

4.8.6 Dry Run

If an environment variable PYLAUNCHER_DRYRUN is set (to any value), the launcher will output the command it would have run, but will not actually launch Python. This may be useful for tools that want to use the launcher to detect and then launch Python directly. Note that the command written to standard output is always encoded using UTF-8, and may not render correctly in the console.

4.8.7 Install on demand

If an environment variable PYLAUNCHER_ALLOW_INSTALL is set (to any value), and the requested Python version is not installed but is available on the Microsoft Store, the launcher will attempt to install it. This may require user interaction to complete, and you may need to run the command again.

An additional PYLAUNCHER_ALWAYS_INSTALL variable causes the launcher to always try to install Python, even if it is detected. This is mainly intended for testing (and should be used with PYLAUNCHER_DRYRUN).

4.8.8 Return codes

The following exit codes may be returned by the Python launcher. Unfortunately, there is no way to distinguish these from the exit code of Python itself.

The names of codes are as used in the sources, and are only for reference. There is no way to access or resolve them apart from reading this page. Entries are listed in alphabetical order of names.

Name	Value	Description
RC_BAD_VENV_CFG	107	A pyvenv.cfg was found but is corrupt.
RC_CREATE_PROCESS	101	Failed to launch Python.
RC_INSTALLING	111	An install was started, but the command will need to be re-run after it com-
		pletes.
RC_INTERNAL_ERROR	109	Unexpected error. Please report a bug.
RC_NO_COMMANDLINI	108	Unable to obtain command line from the operating system.
RC_NO_PYTHON	103	Unable to locate the requested version.
RC_NO_VENV_CFG	106	A pyvenv.cfg was required but not found.

4.9 Finding modules

These notes supplement the description at sys-path-init with detailed Windows notes.

When no ._pth file is found, this is how sys.path is populated on Windows:

- An empty entry is added at the start, which corresponds to the current directory.
- If the environment variable *PYTHONPATH* exists, as described in *Environment variables*, its entries are added next. Note that on Windows, paths in this variable must be separated by semicolons, to distinguish them from the colon used in drive identifiers (C:\ etc.).

- Additional "application paths" can be added in the registry as subkeys of \SOFTWARE\Python\PythonCore{version}\PythonPath under both the HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE hives. Subkeys which have semicolon-delimited path strings as their default value will cause each path to be added to sys.path. (Note that all known installers only use HKLM, so HKCU is typically empty.)
- If the environment variable <code>PYTHONHOME</code> is set, it is assumed as "Python Home". Otherwise, the path of the main Python executable is used to locate a "landmark file" (either <code>Lib\os.py</code> or <code>pythonXY.zip</code>) to deduce the "Python Home". If a Python home is found, the relevant sub-directories added to <code>sys.path(Lib, plat-win, etc)</code> are based on that folder. Otherwise, the core Python path is constructed from the PythonPath stored in the registry.
- If the Python Home cannot be located, no *PYTHONPATH* is specified in the environment, and no registry entries can be found, a default path with relative entries is used (e.g. .\Lib; .\plat-win, etc).

If a pyvenv.cfg file is found alongside the main executable or in the directory one level above the executable, the following variations apply:

• If home is an absolute path and PYTHONHOME is not set, this path is used instead of the path to the main executable when deducing the home location.

The end result of all this is:

- When running python.exe, or any other .exe in the main Python directory (either an installed version, or directly from the PCbuild directory), the core path is deduced, and the core paths in the registry are ignored. Other "application paths" in the registry are always read.
- When Python is hosted in another .exe (different directory, embedded via COM, etc), the "Python Home" will not be deduced, so the core path from the registry is used. Other "application paths" in the registry are always read
- If Python can't find its home and there are no registry value (frozen .exe, some very strange installation setup) you get a path with some default, but relative, paths.

For those who want to bundle Python into their application or distribution, the following advice will prevent conflicts with other installations:

- Include a ._pth file alongside your executable containing the directories to include. This will ignore paths listed in the registry and environment variables, and also ignore site unless import site is listed.
- If you are loading python3.dll or python37.dll in your own executable, explicitly set PyConfig. module_search_paths before Py_InitializeFromConfig().
- Clear and/or overwrite PYTHONPATH and set PYTHONHOME before launching python.exe from your application.
- If you cannot use the previous suggestions (for example, you are a distribution that allows people to run python.exe directly), ensure that the landmark file (Lib\os.py) exists in your install directory. (Note that it will not be detected inside a ZIP file, but a correctly named ZIP file will be detected instead.)

These will ensure that the files in a system-wide installation will not take precedence over the copy of the standard library bundled with your application. Otherwise, your users may experience problems using your application. Note that the first suggestion is the best, as the others may still be susceptible to non-standard paths in the registry and user site-packages.

Changed in version 3.6: Add ._pth file support and removes applocal option from pyvenv.cfg.

Changed in version 3.6: Add python XX. zip as a potential landmark when directly adjacent to the executable.

Deprecated since version 3.6: Modules specified in the registry under Modules (not PythonPath) may be imported by importlib.machinery.WindowsRegistryFinder. This finder is enabled on Windows in 3.6.0 and earlier, but may need to be explicitly added to sys.meta_path in the future.

4.10 Additional modules

Even though Python aims to be portable among all platforms, there are features that are unique to Windows. A couple of modules, both in the standard library and external, and snippets exist to use these features.

The Windows-specific standard modules are documented in mswin-specific-services.

4.10.1 PyWin32

The PyWin32 module by Mark Hammond is a collection of modules for advanced Windows-specific support. This includes utilities for:

- Component Object Model (COM)
- Win32 API calls
- Registry
- Event log
- Microsoft Foundation Classes (MFC) user interfaces

PythonWin is a sample MFC application shipped with PyWin32. It is an embeddable IDE with a built-in debugger.



Win32 How Do I...?

by Tim Golden

Python and COM

by David and Paul Boddie

4.10.2 cx_Freeze

cx_Freeze wraps Python scripts into executable Windows programs (\star .exe files). When you have done this, you can distribute your application without requiring your users to install Python.

4.11 Compiling Python on Windows

If you want to compile CPython yourself, first thing you should do is get the source. You can download either the latest release's source or just grab a fresh checkout.

The source tree contains a build solution and project files for Microsoft Visual Studio, which is the compiler used to build the official Python releases. These files are in the PCbuild directory.

Check PCbuild/readme.txt for general information on the build process.

For extension modules, consult building-on-windows.

4.12 Other Platforms

With ongoing development of Python, some platforms that used to be supported earlier are no longer supported (due to the lack of users or developers). Check **PEP 11** for details on all unsupported platforms.

- Windows CE is no longer supported since Python 3 (if it ever was).
- The Cygwin installer offers to install the Python interpreter as well

See Python for Windows for detailed information about platforms with pre-compiled installers.

USING PYTHON ON MACOS

This document aims to give an overview of macOS-specific behavior you should know about to get started with Python on Mac computers. Python on a Mac running macOS is very similar to Python on other Unix-derived platforms, but there are some differences in installation and some features.

There are various ways to obtain and install Python for macOS. Pre-built versions of the most recent versions of Python are available from a number of distributors. Much of this document describes use of the Pythons provided by the CPython release team for download from the python.org website. See *Alternative Distributions* for some other options.

5.1 Using Python for macOS from python.org

5.1.1 Installation steps

For current Python versions (other than those in security status), the release team produces a **Python for macOS** installer package for each new release. A list of available installers is available here. We recommend using the most recent supported Python version where possible. Current installers provide a universal2 binary build of Python which runs natively on all Macs (Apple Silicon and Intel) that are supported by a wide range of macOS versions, currently typically from at least **macOS 10.13 High Sierra** on.

The downloaded file is a standard macOS installer package file (.pkg). File integrity information (checksum, size, sigstore signature, etc) for each file is included on the release download page. Installer packages and their contents are signed and notarized with Python Software Foundation Apple Developer ID certificates to meet macOS Gatekeeper requirements.

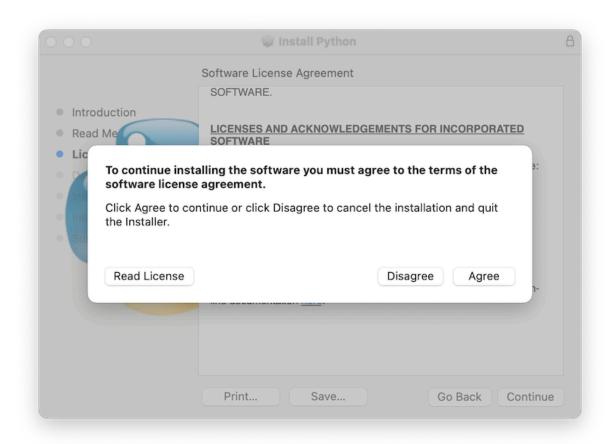
For a default installation, double-click on the downloaded installer package file. This should launch the standard macOS Installer app and display the first of several installer windows steps.



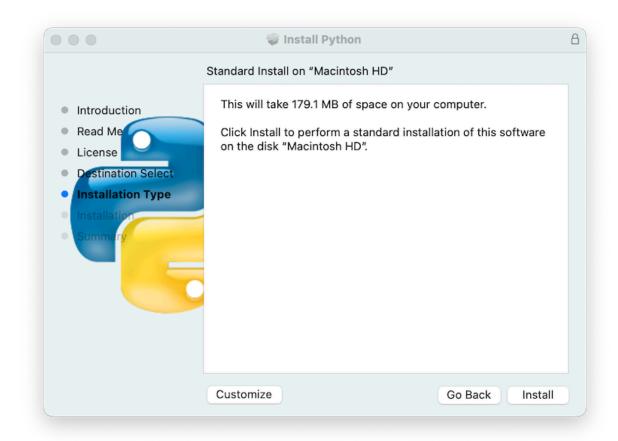
Clicking on the **Continue** button brings up the **Read Me** for this installer. Besides other important information, the **Read Me** documents which Python version is going to be installed and on what versions of macOS it is supported. You may need to scroll through to read the whole file. By default, this **Read Me** will also be installed in /Applications/Python 3.13/ and available to read anytime.



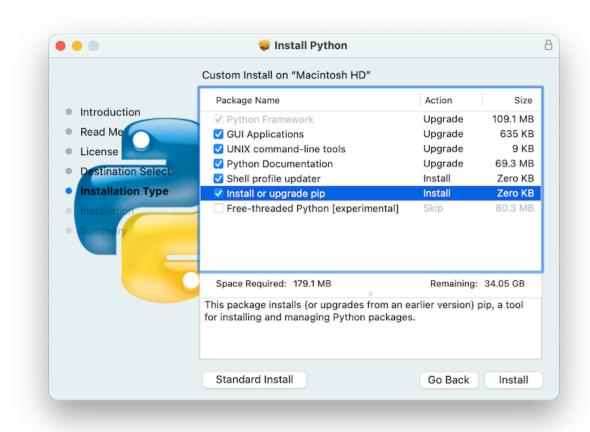
Clicking on **Continue** proceeds to display the license for Python and for other included software. You will then need to **Agree** to the license terms before proceeding to the next step. This license file will also be installed and available to be read later.



After the license terms are accepted, the next step is the **Installation Type** display. For most uses, the standard set of installation operations is appropriate.



By pressing the **Customize** button, you can choose to omit or select certain package components of the installer. Click on each package name to see a description of what it installs. To also install support for the optional experimental free-threaded feature, see *Installing Free-threaded Binaries*.

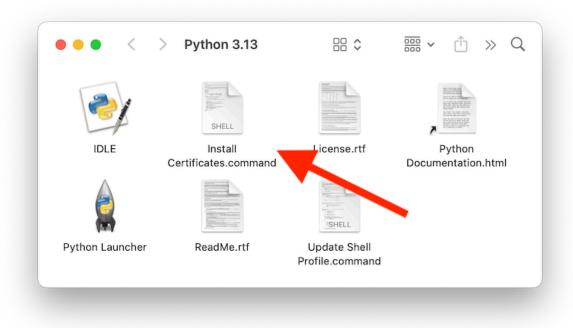


In either case, clicking **Install** will begin the install process by asking permission to install new software. A macOS user name with Administrator privilege is needed as the installed Python will be available to all users of the Mac.

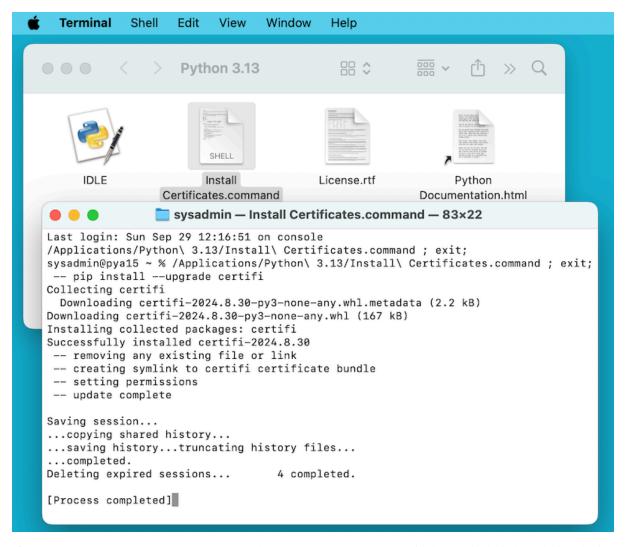
When the installation is complete, the **Summary** window will appear.



Double-click on the Install Certificates.command icon or file in the /Applications/Python 3.13/ window to complete the installation.



This will open a temporary **Terminal** shell window that will use the new Python to download and install SSL root certificates for its use.



If Successfully installed certifiand update complete appears in the terminal window, the installation is complete. Close this terminal window and the installer window.

A default install will include:

- A Python 3.13 folder in your Applications folder. In here you find IDLE, the development environment that is a standard part of official Python distributions; and Python Launcher, which handles double-clicking Python scripts from the macOS Finder.
- A framework /Library/Frameworks/Python.framework, which includes the Python executable and libraries. The installer adds this location to your shell path. To uninstall Python, you can remove these three things. Symlinks to the Python executable are placed in /usr/local/bin/.

1 Note

Recent versions of macOS include a python3 command in /usr/bin/python3 that links to a usually older and incomplete version of Python provided by and for use by the Apple development tools, xcode or the Command Line Tools for Xcode. You should never modify or attempt to delete this installation, as it is Apple-controlled and is used by Apple-provided or third-party software. If you choose to install a newer Python version from python.org, you will have two different but functional Python installations on your computer that can co-exist. The default installer options should ensure that its python3 will be used instead of the system python3.

5.1.2 How to run a Python script

There are two ways to invoke the Python interpreter. If you are familiar with using a Unix shell in a terminal window, you can invoke python3.13 or python3 optionally followed by one or more command line options (described in *Command line and environment*). The Python tutorial also has a useful section on using Python interactively from a shell.

You can also invoke the interpreter through an integrated development environment. idle is a basic editor and interpreter environment which is included with the standard distribution of Python. **IDLE** includes a Help menu that allows you to access Python documentation. If you are completely new to Python, you can read the tutorial introduction in that document.

There are many other editors and IDEs available, see *Editors and IDEs* for more information.

To run a Python script file from the terminal window, you can invoke the interpreter with the name of the script file:

```
python3.13 myscript.py
```

To run your script from the Finder, you can either:

- Drag it to Python Launcher.
- Select Python Launcher as the default application to open your script (or any .py script) through the Finder
 Info window and double-click it. Python Launcher has various preferences to control how your script is
 launched. Option-dragging allows you to change these for one invocation, or use its Preferences menu to
 change things globally.

Be aware that running the script directly from the macOS Finder might produce different results than when running from a terminal window as the script will not be run in the usual shell environment including any setting of environment variables in shell profiles. And, as with any other script or program, be certain of what you are about to run.

5.2 Alternative Distributions

Besides the standard python.org for macOS installer, there are third-party distributions for macOS that may include additional functionality. Some popular distributions and their key features:

ActivePython

Installer with multi-platform compatibility, documentation

Anaconda

Popular scientific modules (such as numpy, scipy, and pandas) and the conda package manager.

Homebrey

Package manager for macOS including multiple versions of Python and many third-party Python-based packages (including numpy, scipy, and pandas).

MacPorts

Another package manager for macOS including multiple versions of Python and many third-party Python-based packages. May include pre-built versions of Python and many packages for older versions of macOS.

Note that distributions might not include the latest versions of Python or other libraries, and are not maintained or supported by the core Python team.

5.3 Installing Additional Python Packages

Refer to the Python Packaging User Guide for more information.

5.4 GUI Programming

There are several options for building GUI applications on the Mac with Python.

The standard Python GUI toolkit is tkinter, based on the cross-platform Tk toolkit (https://www.tcl.tk). A macOSnative version of Tk is included with the installer.

PyObjC is a Python binding to Apple's Objective-C/Cocoa framework. Information on PyObjC is available from pyobjc.

A number of alternative macOS GUI toolkits are available including:

- PySide: Official Python bindings to the Qt GUI toolkit.
- PyQt: Alternative Python bindings to Qt.
- Kivy: A cross-platform GUI toolkit that supports desktop and mobile platforms.
- Toga: Part of the BeeWare Project; supports desktop, mobile, web and console apps.
- wxPython: A cross-platform toolkit that supports desktop operating systems.

5.5 Advanced Topics

5.5.1 Installing Free-threaded Binaries

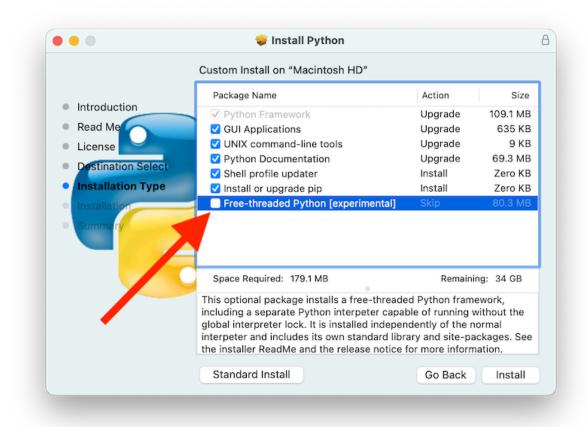
Added in version 3.13: (Experimental)



Everything described in this section is considered experimental, and should be expected to change in future releases.

The python.org Python for macOS installer package can optionally install an additional build of Python 3.13 that supports PEP 703, the experimental free-threading feature (running with the global interpreter lock disabled). Check the release page on python.org for possible updated information.

Because this feature is still considered experimental, the support for it is not installed by default. It is packaged as a separate install option, available by clicking the Customize button on the Installation Type step of the installer as described above.



If the box next to the **Free-threaded Python** package name is checked, a separate PythonT.framework will also be installed alongside the normal Python.framework in /Library/Frameworks. This configuration allows a free-threaded Python 3.13 build to co-exist on your system with a traditional (GIL only) Python 3.13 build with minimal risk while installing or testing. This installation layout is itself experimental and is subject to change in future releases.

Known cautions and limitations:

- The UNIX command-line tools package, which is selected by default, will install links in /usr/local/bin for python3.13t, the free-threaded interpreter, and python3.13t-config, a configuration utility which may be useful for package builders. Since /usr/local/bin is typically included in your shell PATH, in most cases no changes to your PATH environment variables should be needed to use python3.13t.
- For this release, the Shell profile updater package and the Update Shell Profile.command in / Applications/Python 3.13/ do not support the free-threaded package.
- The free-threaded build and the traditional build have separate search paths and separate site-packages directories so, by default, if you need a package available in both builds, it may need to be installed in both. The free-threaded package will install a separate instance of pip for use with python3.13t.
 - To install a package using pip without a venv:

```
python3.13t -m pip install <package_name>
```

• When working with multiple Python environments, it is usually safest and easiest to create and use virtual environments. This can avoid possible command name conflicts and confusion about which Python is in use:

```
python3.13t -m venv <venv_name>
```

then activate.

• To run a free-threaded version of IDLE:

```
python3.13t -m idlelib
```

- The interpreters in both builds respond to the same *PYTHON environment variables* which may have unexpected results, for example, if you have PYTHONPATH set in a shell profile. If necessary, there are *command line options* like -E to ignore these environment variables.
- The free-threaded build links to the third-party shared libraries, such as OpenSSL and Tk, installed in the traditional framework. This means that both builds also share one set of trust certificates as installed by the Install Certificates.command script, thus it only needs to be run once.
- If you cannot depend on the link in /usr/local/bin pointing to the python.org free-threaded python3.

 13t (for example, if you want to install your own version there or some other distribution does), you can explicitly set your shell PATH environment variable to include the PythonT framework bin directory:

```
export PATH="/Library/Frameworks/PythonT.framework/Versions/3.13/bin":"$PATH"
```

The traditional framework installation by default does something similar, except for Python.framework. Be aware that having both framework bin directories in PATH can lead to confusion if there are duplicate names like python3.13 in both; which one is actually used depends on the order they appear in PATH. The which python3.x or which python3.xt commands can show which path is being used. Using virtual environments can help avoid such ambiguities. Another option might be to create a shell alias to the desired interpreter, like:

```
alias py3.13="/Library/Frameworks/Python.framework/Versions/3.13/bin/python3.13

→"
alias py3.13t="/Library/Frameworks/PythonT.framework/Versions/3.13/bin/python3.

→13t"
```

5.5.2 Installing using the command line

If you want to use automation to install the python.org installer package (rather than by using the familiar macOS Installer GUI app), the macOS command line installer utility lets you select non-default options, too. If you are not familiar with installer, it can be somewhat cryptic (see man installer for more information). As an example, the following shell snippet shows one way to do it, using the 3.13.0b2 release and selecting the free-threaded interpreter option:

```
RELEASE="python-3.13.0b2-macos11.pkg"
# download installer pkg
curl -O https://www.python.org/ftp/python/3.13.0/${RELEASE}
# create installer choicechanges to customize the install:
  enable the PythonTFramework-3.13 package
   while accepting the other defaults (install all other packages)
cat > ./choicechanges.plist <<EOF</pre>
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/</pre>
→PropertyList-1.0.dtd">
<plist version="1.0">
<array>
        <dict>
                <key>attributeSetting</key>
                <integer>1</integer>
                <key>choiceAttribute</key>
                <string>selected</string>
                <key>choiceIdentifier</key>
                <string>org.python.Python.PythonTFramework-3.13</string>
        </dict>
</array>
</plist>
EOF
```

(continues on next page)

(continued from previous page)

```
sudo installer -pkg ./${RELEASE}$ -applyChoiceChangesXML ./choicechanges.plist - \totarget /
```

You can then test that both installer builds are now available with something like:

```
💲 # test that the free-threaded interpreter was installed if the Unix Command...
→ Tools package was enabled
$ /usr/local/bin/python3.13t -VV
Python 3.13.0b2 experimental free-threading build (v3.13.0b2:3a83b172af, Jun 5_
→2024, 12:57:31) [Clang 15.0.0 (clang-1500.3.9.4)]
      and the traditional interpreter
$ /usr/local/bin/python3.13 -VV
Python 3.13.0b2 (v3.13.0b2:3a83b172af, Jun 5 2024, 12:50:24) [Clang 15.0.0 (clang-
→1500.3.9.4)]
$ # test that they are also available without the prefix if /usr/local/bin is on
→$PATH
$ python3.13t -VV
Python 3.13.0b2 experimental free-threading build (v3.13.0b2:3a83b172af, Jun 5_
→2024, 12:57:31) [Clang 15.0.0 (clang-1500.3.9.4)]
$ python3.13 -VV
Python 3.13.0b2 (v3.13.0b2:3a83b172af, Jun 5 2024, 12:50:24) [Clang 15.0.0 (clang-
→1500.3.9.4)]
```

1 Note

Current python.org installers only install to fixed locations like /Library/Frameworks/, /Applications, and /usr/local/bin. You cannot use the installer -domain option to install to other locations.

5.5.3 Distributing Python Applications

A range of tools exist for converting your Python code into a standalone distributable application:

- py2app: Supports creating macOS .app bundles from a Python project.
- Briefcase: Part of the BeeWare Project; a cross-platform packaging tool that supports creation of .app bundles on macOS, as well as managing signing and notarization.
- PyInstaller: A cross-platform packaging tool that creates a single file or folder as a distributable artifact.

5.5.4 App Store Compliance

Apps submitted for distribution through the macOS App Store must pass Apple's app review process. This process includes a set of automated validation rules that inspect the submitted application bundle for problematic code.

The Python standard library contains some code that is known to violate these automated rules. While these violations appear to be false positives, Apple's review rules cannot be challenged. Therefore, it is necessary to modify the Python standard library for an app to pass App Store review.

The Python source tree contains a patch file that will remove all code that is known to cause issues with the App Store review process. This patch is applied automatically when CPython is configured with the --with-app-store-compliance option.

This patch is not normally required to use CPython on a Mac; nor is it required if you are distributing an app *outside* the macOS App Store. It is *only* required if you are using the macOS App Store as a distribution channel.

5.6 Other Resources

The python.org Help page has links to many useful resources. The Pythonmac-SIG mailing list is another support resource specifically for Python users and developers on the Mac.

USING PYTHON ON ANDROID

Python on Android is unlike Python on desktop platforms. On a desktop platform, Python is generally installed as a system resource that can be used by any user of that computer. Users then interact with Python by running a **python** executable and entering commands at an interactive prompt, or by running a Python script.

On Android, there is no concept of installing as a system resource. The only unit of software distribution is an "app". There is also no console where you could run a **python** executable, or interact with a Python REPL.

As a result, the only way you can use Python on Android is in embedded mode – that is, by writing a native Android application, embedding a Python interpreter using <code>libpython</code>, and invoking Python code using the Python embedding API. The full Python interpreter, the standard library, and all your Python code is then packaged into your app for its own private use.

The Python standard library has some notable omissions and restrictions on Android. See the API availability guide for details.

6.1 Adding Python to an Android app

These instructions are only needed if you're planning to compile Python for Android yourself. Most users should *not* need to do this. Instead, use one of the following tools, which will provide a much easier experience:

- Briefcase, from the BeeWare project
- Buildozer, from the Kivy project
- Chaquopy
- pyqtdeploy
- Termux

If you're sure you want to do all of this manually, read on. You can use the testbed app as a guide; each step below contains a link to the relevant file.

- $\bullet\,$ Build Python by following the instructions in Android/README.md.
- Add code to your build.gradle file to copy the following items into your project. All except your own Python code can be copied from cross-build/HOST/prefix/lib:
 - In your JNI libraries:
 - * libpython*.*.so
 - * lib*_python.so (external libraries such as OpenSSL)
 - In your assets:
 - * python*.* (the Python standard library)
 - * python*.*/site-packages (your own Python code)
- Add code to your app to extract the assets to the filesystem.
- Add code to your app to start Python in embedded mode. This will need to be C code called via JNI.

USING PYTHON ON IOS

Authors

Russell Keith-Magee (2024-03)

Python on iOS is unlike Python on desktop platforms. On a desktop platform, Python is generally installed as a system resource that can be used by any user of that computer. Users then interact with Python by running a **python** executable and entering commands at an interactive prompt, or by running a Python script.

On iOS, there is no concept of installing as a system resource. The only unit of software distribution is an "app". There is also no console where you could run a **python** executable, or interact with a Python REPL.

As a result, the only way you can use Python on iOS is in embedded mode - that is, by writing a native iOS application, and embedding a Python interpreter using libPython, and invoking Python code using the Python embedding API. The full Python interpreter, the standard library, and all your Python code is then packaged as a standalone bundle that can be distributed via the iOS App Store.

If you're looking to experiment for the first time with writing an iOS app in Python, projects such as BeeWare and Kivy will provide a much more approachable user experience. These projects manage the complexities associated with getting an iOS project running, so you only need to deal with the Python code itself.

7.1 Python at runtime on iOS

7.1.1 iOS version compatibility

The minimum supported iOS version is specified at compile time, using the —host option to configure. By default, when compiled for iOS, Python will be compiled with a minimum supported iOS version of 13.0. To use a different minimum iOS version, provide the version number as part of the —host argument - for example, —host=arm64-apple-ios15.4-simulator would compile an ARM64 simulator build with a deployment target of 15.4.

7.1.2 Platform identification

When executing on iOS, sys.platform will report as ios. This value will be returned on an iPhone or iPad, regardless of whether the app is running on the simulator or a physical device.

Information about the specific runtime environment, including the iOS version, device model, and whether the device is a simulator, can be obtained using platform.ios_ver().platform.system() will report iOS or iPadOS, depending on the device.

os.uname() reports kernel-level details; it will report a name of Darwin.

7.1.3 Standard library availability

The Python standard library has some notable omissions and restrictions on iOS. See the API availability guide for iOS for details.

7.1.4 Binary extension modules

One notable difference about iOS as a platform is that App Store distribution imposes hard requirements on the packaging of an application. One of these requirements governs how binary extension modules are distributed.

The iOS App Store requires that *all* binary modules in an iOS app must be dynamic libraries, contained in a framework with appropriate metadata, stored in the Frameworks folder of the packaged app. There can be only a single binary per framework, and there can be no executable binary material outside the Frameworks folder.

This conflicts with the usual Python approach for distributing binaries, which allows a binary extension module to be loaded from any location on <code>sys.path</code>. To ensure compliance with App Store policies, an iOS project must post-process any Python packages, converting <code>.so</code> binary modules into individual standalone frameworks with appropriate metadata and signing. For details on how to perform this post-processing, see the guide for *adding Python to your project*.

To help Python discover binaries in their new location, the original .so file on sys.path is replaced with a .fwork file. This file is a text file containing the location of the framework binary, relative to the app bundle. To allow the framework to resolve back to the original location, the framework must contain a .origin file that contains the location of the .fwork file, relative to the app bundle.

For example, consider the case of an import from foo.bar import _whiz, where _whiz is implemented with the binary module sources/foo/bar/_whiz.abi3.so, with sources being the location registered on sys. path, relative to the application bundle. This module *must* be distributed as Frameworks/foo.bar._whiz. framework/foo.bar._whiz (creating the framework name from the full import path of the module), with an Info.plist file in the .framework directory identifying the binary as a framework. The foo.bar._whiz module would be represented in the original location with a sources/foo/bar/_whiz.abi3.fwork marker file, containing the path Frameworks/foo.bar._whiz/foo.bar._whiz. The framework would also contain Frameworks/foo.bar._whiz.framework/foo.bar._whiz.origin, containing the path to the .fwork file.

When running on iOS, the Python interpreter will install an AppleFrameworkLoader that is able to read and import .fwork files. Once imported, the __file__ attribute of the binary module will report as the location of the .fwork file. However, the ModuleSpec for the loaded module will report the origin as the location of the binary in the framework folder.

7.1.5 Compiler stub binaries

Xcode doesn't expose explicit compilers for iOS; instead, it uses an xcrun script that resolves to a full compiler path (e.g., xcrun --sdk iphoneos clang to get the clang for an iPhone device). However, using this script poses two problems:

- The output of xcrun includes paths that are machine specific, resulting in a sysconfig module that cannot be shared between users; and
- It results in CC/CPP/LD/AR definitions that include spaces. There is a lot of C ecosystem tooling that assumes that you can split a command line at the first space to get the path to the compiler executable; this isn't the case when using xcrun.

To avoid these problems, Python provided stubs for these tools. These stubs are shell script wrappers around the underingly xcrun tools, distributed in a bin folder distributed alongside the compiled iOS framework. These scripts are relocatable, and will always resolve to the appropriate local system paths. By including these scripts in the bin folder that accompanies a framework, the contents of the sysconfig module becomes useful for end-users to compile their own modules. When compiling third-party Python modules for iOS, you should ensure these stub binaries are on your path.

7.2 Installing Python on iOS

7.2.1 Tools for building iOS apps

Building for iOS requires the use of Apple's Xcode tooling. It is strongly recommended that you use the most recent stable release of Xcode. This will require the use of the most (or second-most) recently released macOS version, as Apple does not maintain Xcode for older macOS versions. The Xcode Command Line Tools are not sufficient for iOS development; you need a *full* Xcode install.

If you want to run your code on the iOS simulator, you'll also need to install an iOS Simulator Platform. You should be prompted to select an iOS Simulator Platform when you first run Xcode. Alternatively, you can add an iOS Simulator Platform by selecting from the Platforms tab of the Xcode Settings panel.

7.2.2 Adding Python to an iOS project

Python can be added to any iOS project, using either Swift or Objective C. The following examples will use Objective C; if you are using Swift, you may find a library like PythonKit to be helpful.

To add Python to an iOS Xcode project:

- 1. Build or obtain a Python XCFramework. See the instructions in iOS/README.rst (in the CPython source distribution) for details on how to build a Python XCFramework. At a minimum, you will need a build that supports arm64-apple-ios, plus one of either arm64-apple-ios-simulator or x86_64-apple-ios-simulator.
- 2. Drag the XCframework into your iOS project. In the following instructions, we'll assume you've dropped the XCframework into the root of your project; however, you can use any other location that you want by adjusting paths as needed.
- 3. Drag the iOS/Resources/dylib-Info-template.plist file into your project, and ensure it is associated with the app target.
- 4. Add your application code as a folder in your Xcode project. In the following instructions, we'll assume that your user code is in a folder named app in the root of your project; you can use any other location by adjusting paths as needed. Ensure that this folder is associated with your app target.
- 5. Select the app target by selecting the root node of your Xcode project, then the target name in the sidebar that appears.
- 6. In the "General" settings, under "Frameworks, Libraries and Embedded Content", add Python. xcframework, with "Embed & Sign" selected.
- 7. In the "Build Settings" tab, modify the following:
 - · Build Options
 - User Script Sandboxing: No
 - Enable Testability: Yes
 - · Search Paths
 - Framework Search Paths: \$ (PROJECT_DIR)
 - Header Search Paths: "\$ (BUILT_PRODUCTS_DIR) / Python.framework/Headers"
 - Apple Clang Warnings All languages
 - Quoted Include In Framework Header: No
- 8. Add a build step that copies the Python standard library into your app. In the "Build Phases" tab, add a new "Run Script" build step *before* the "Embed Frameworks" step, but *after* the "Copy Bundle Resources" step. Name the step "Install Target Specific Python Standard Library", disable the "Based on dependency analysis" checkbox, and set the script content to:

```
set -e

mkdir -p "$CODESIGNING_FOLDER_PATH/python/lib"

if [ "$EFFECTIVE_PLATFORM_NAME" = "-iphonesimulator" ]; then
    echo "Installing Python modules for iOS Simulator"
    rsync -au --delete "$PROJECT_DIR/Python.xcframework/ios-arm64_x86_64-
    simulator/lib/" "$CODESIGNING_FOLDER_PATH/python/lib/"

else
    echo "Installing Python modules for iOS Device"
    rsync -au --delete "$PROJECT_DIR/Python.xcframework/ios-arm64/lib/" "
```

(continues on next page)

(continued from previous page)

```
→$CODESIGNING_FOLDER_PATH/python/lib/"

fi
```

Note that the name of the simulator "slice" in the XCframework may be different, depending the CPU architectures your XCFramework supports.

9. Add a second build step that processes the binary extension modules in the standard library into "Framework" format. Add a "Run Script" build step *directly after* the one you added in step 8, named "Prepare Python Binary Modules". It should also have "Based on dependency analysis" unchecked, with the following script content:

```
set -e
install_dylib () {
   INSTALL BASE=$1
   FULL_EXT=$2
    # The name of the extension file
   EXT=$ (basename "$FULL_EXT")
    # The location of the extension file, relative to the bundle
   RELATIVE_EXT=${FULL_EXT#$CODESIGNING_FOLDER_PATH/}
    # The path to the extension file, relative to the install base
   PYTHON_EXT=${RELATIVE_EXT/$INSTALL_BASE/}
    # The full dotted name of the extension module, constructed from the file.
\rightarrow path.
   FULL_MODULE_NAME=$(echo $PYTHON_EXT | cut -d "." -f 1 | tr "/" ".");
    # A bundle identifier; not actually used, but required by Xcode framework_
   FRAMEWORK_BUNDLE_ID=$ (echo $PRODUCT_BUNDLE_IDENTIFIER.$FULL_MODULE_NAME |_
→tr "_" "-")
    # The name of the framework folder.
   FRAMEWORK_FOLDER="Frameworks/$FULL_MODULE_NAME.framework"
    # If the framework folder doesn't exist, create it.
   if [ ! -d "$CODESIGNING_FOLDER_PATH/$FRAMEWORK_FOLDER" ]; then
        echo "Creating framework for $RELATIVE_EXT"
       mkdir -p "$CODESIGNING_FOLDER_PATH/$FRAMEWORK_FOLDER"
       cp "$CODESIGNING_FOLDER_PATH/dylib-Info-template.plist" "$CODESIGNING_
→FOLDER_PATH/$FRAMEWORK_FOLDER/Info.plist"
       plutil -replace CFBundleExecutable -string "$FULL_MODULE_NAME" "
→$CODESIGNING_FOLDER_PATH/$FRAMEWORK_FOLDER/Info.plist"
       plutil -replace CFBundleIdentifier -string "$FRAMEWORK_BUNDLE_ID" "
→$CODESIGNING_FOLDER_PATH/$FRAMEWORK_FOLDER/Info.plist"
   fi
   echo "Installing binary for $FRAMEWORK FOLDER/$FULL MODULE NAME"
   mv "$FULL_EXT" "$CODESIGNING_FOLDER_PATH/$FRAMEWORK_FOLDER/$FULL_MODULE_
→NAME"
   # Create a placeholder .fwork file where the .so was
   echo "$FRAMEWORK_FOLDER/$FULL_MODULE_NAME" > ${FULL_EXT%.so}.fwork
   # Create a back reference to the .so file location in the framework
   echo "${RELATIVE_EXT%.so}.fwork" > "$CODESIGNING_FOLDER_PATH/$FRAMEWORK_
→FOLDER/$FULL_MODULE_NAME.origin"
PYTHON_VER=$(ls -1 "$CODESIGNING_FOLDER_PATH/python/lib")
echo "Install Python $PYTHON_VER standard library extension modules..."
find "$CODESIGNING_FOLDER_PATH/python/lib/$PYTHON_VER/lib-dynload" -name "*.so
```

(continues on next page)

(continued from previous page)

```
→" | while read FULL_EXT; do
    install_dylib python/lib/$PYTHON_VER/lib-dynload/ "$FULL_EXT"
done

# Clean up dylib template
rm -f "$CODESIGNING_FOLDER_PATH/dylib-Info-template.plist"

echo "Signing frameworks as $EXPANDED_CODE_SIGN_IDENTITY_NAME ($EXPANDED_CODE_
    →SIGN_IDENTITY)..."
find "$CODESIGNING_FOLDER_PATH/Frameworks" -name "*.framework" -exec /usr/bin/
    →codesign --force --sign "$EXPANDED_CODE_SIGN_IDENTITY" ${OTHER_CODE_SIGN_
    →FLAGS:-} -o runtime --timestamp=none --preserve-metadata=identifier,
    →entitlements,flags --generate-entitlement-der "{}" \;
```

- 10. Add Objective C code to initialize and use a Python interpreter in embedded mode. You should ensure that:
 - UTF-8 mode is enabled;
 - Buffered stdio is disabled;
 - Writing bytecode is disabled;
 - Signal handlers are enabled;
 - PYTHONHOME for the interpreter is configured to point at the python subfolder of your app's bundle; and
 - The PYTHONPATH for the interpreter includes:
 - the python/lib/python3.X subfolder of your app's bundle,
 - the python/lib/python3.X/lib-dynload subfolder of your app's bundle, and
 - the app subfolder of your app's bundle

Your app's bundle location can be determined using [[NSBundle mainBundle] resourcePath].

Steps 8, 9 and 10 of these instructions assume that you have a single folder of pure Python application code, named app. If you have third-party binary modules in your app, some additional steps will be required:

- You need to ensure that any folders containing third-party binaries are either associated with the app target, or copied in as part of step 8. Step 8 should also purge any binaries that are not appropriate for the platform a specific build is targeting (i.e., delete any device binaries if you're building an app targeting the simulator).
- Any folders that contain third-party binaries must be processed into framework form by step 9. The invocation of install_dylib that processes the lib-dynload folder can be copied and adapted for this purpose.
- If you're using a separate folder for third-party packages, ensure that folder is included as part of the PYTHONPATH configuration in step 10.

7.3 App Store Compliance

The only mechanism for distributing apps to third-party iOS devices is to submit the app to the iOS App Store; apps submitted for distribution must pass Apple's app review process. This process includes a set of automated validation rules that inspect the submitted application bundle for problematic code.

The Python standard library contains some code that is known to violate these automated rules. While these violations appear to be false positives, Apple's review rules cannot be challenged; so, it is necessary to modify the Python standard library for an app to pass App Store review.

The Python source tree contains a patch file that will remove all code that is known to cause issues with the App Store review process. This patch is applied automatically when building for iOS.

CHAPTER

EIGHT

EDITORS AND IDES

There are a number of IDEs that support Python programming language. Many editors and IDEs provide syntax highlighting, debugging tools, and PEP 8 checks.

Please go to Python Editors and Integrated Development Environments for a comprehensive list.

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, for example using setattr(), if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with getattr().

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 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 = $\{n: n ** 2 \text{ for } n \text{ in range (10)} \}$ generates a dictionary containing key n mapped to value n ** 2. See comprehensions.

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 <code>codeobject.co_freevars</code> 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 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.

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 <code>dict</code>) 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 --disable-gil build configuration. After building Python with this option, code must be run with -X gil=0 or after setting the $PYTHON_GIL=0$ 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 pyc

A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See pyc-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. Module Spec.

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 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 comprehensions.

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 specialnames.

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:

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

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 annassign.

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	callback, 87
, 85	CC
-?	command line option, 27
	C-contiguous, 88
command line option, 5	CFLAGS, 30, 39, 40
%APPDATA%, 49	command line option, 27
>>>, 85	CFLAGS_NODIST, 39, 40
future, 91	check-hash-based-pycs
slots, 98	command line option, 6
A	class, 87
	class variable, 87
abstract base class, 85	closure variable, 87
annotation, 85	command line option
argument, 85	-?, 5
asynchronous context manager, 86	-в, 6
asynchronous generator, 86	-b, 6
asynchronous generator iterator, 86	•
asynchronous iterable, 86	BOLT_APPLY_FLAGS, 30 BOLT_INSTRUMENT_FLAGS, 30
asynchronous iterator, 86	
attribute, 86	build, 35
awaitable, 86	BZIP2_CFLAGS, 27
П	BZIP2_LIBS, 27
В	-c, 3
-B	cc, 27
command line option, 6	CFLAGS, 27
-b	check-hash-based-pycs, 6
command line option, 6	CONFIG_SITE, 36
BDFL, 86	CPP, 27
binary file, 86	CPPFLAGS, 27
BOLT_APPLY_FLAGS	CURSES_CFLAGS, 27
command line option, 30	CURSES_LIBS, 27
BOLT_INSTRUMENT_FLAGS	-d, 6
command line option, 30	disable-gil, 26
borrowed reference, 86	disable-ipv6,24
build	disable-test-modules, 29
command line option, 35	-E, 6
bytecode, 87	enable-big-digits, 24
bytes-like object, 87	enable-bolt, 30
BZIP2_CFLAGS	enable-experimental-jit,27
command line option, 27	enable-framework, 35
BZIP2_LIBS	enable-loadable-sqlite-extensions,
command line option, 27	24
Command Time Operon, 27	enable-optimizations, 29
C	enable-profiling, 31
	enable-pystats, 25
-c	enable-shared, 33
command line option, 3	enable-universalsdk, 34
callable, 87	

```
--enable-wasm-dynamic-linking, 29
                                               --with-build-python, 36
--enable-wasm-pthreads, 29
                                               --with-builtin-hashlib-hashes, 34
--exec-prefix, 29
                                               --with-computed-gotos, 31
GDBM_CFLAGS, 27
                                               --with-dbmliborder, 25
GDBM_LIBS, 28
                                               --with-dtrace, 32
-h.5
                                               --with-emscripten-target, 29
--help, 5
                                               --with-ensurepip, 29
--help-all, 5
                                               --with-framework-name, 35
--help-env, 5
                                               --with-hash-algorithm, 34
--help-xoptions, 5
                                               --with-libc, 33
--host, 35
                                               --with-libm, 33
HOSTRUNNER, 36
                                               --with-libs, 33
-1,6
                                               --with-lto, 30
-i, 6
                                               --with-memory-sanitizer, 32
-J, 10
                                               --with-openss1,33
LDFLAGS, 27
                                               --with-openssl-rpath, 34
LIBB2_CFLAGS, 28
                                               --without-c-locale-coercion, 25
LIBB2_LIBS, 28
                                               --without-decimal-contextvar, 24
LIBEDIT_CFLAGS, 28
                                               --without-doc-strings, 31
LIBEDIT_LIBS, 28
                                               --without-freelists, 25
LIBFFI_CFLAGS, 28
                                               --without-mimalloc, 31
LIBFFI_LIBS, 28
                                               --without-pymalloc, 31
LIBLZMA_CFLAGS, 28
                                               --without-readline, 33
LIBLZMA_LIBS, 28
                                               --without-static-libpython, 33
LIBMPDEC_CFLAGS, 28
                                               --with-pkg-config, 25
LIBMPDEC_LIBS, 28
                                               --with-platlibdir, 25
LIBREADLINE_CFLAGS, 28
                                               --with-pydebug, 32
LIBREADLINE_LIBS, 28
                                               --with-readline, 33
LIBS, 27
                                               --with-ssl-default-suites, 34
LIBSQLITE3_CFLAGS, 28
                                               --with-strict-overflow, 31
LIBSQLITE3_LIBS, 28
                                               --with-suffix, 24
LIBUUID_CFLAGS, 28
                                               --with-system-expat, 33
LIBUUID_LIBS, 28
                                               --with-system-libmpdec, 33
-m, 3
                                               --with-thread-sanitizer, 32
MACHDEP, 27
                                               --with-trace-refs, 32
-0, 6
                                               --with-tzpath, 24
-00,6
                                               --with-undefined-behavior-sanitizer,
-P.6
                                                   32
PANEL CFLAGS, 28
                                               --with-universal-archs, 35
PANEL_LIBS, 28
                                               --with-valgrind, 32
PKG_CONFIG, 27
                                               --with-wheel-pkg-dir, 25
PKG_CONFIG_LIBDIR, 27
                                               -x, 8
PKG_CONFIG_PATH, 27
                                               -x, 8
--prefix, 29
                                               ZLIB_CFLAGS, 28
                                               ZLIB_LIBS, 29
-q, 7
-R, 7
                                           complex number, 88
-s, 7
                                           CONFIG_SITE
-s, 7
                                               command line option, 36
TCLTK_CFLAGS, 28
                                           context, 88
TCLTK_LIBS, 28
                                           context management protocol, 88
-u, 7
                                           context manager, 88
-v, 5
                                           context variable, 88
-v, 7
                                           contiguous, 88
--version, 5
                                           coroutine, 88
-w, 7
                                           coroutine function, 88
--with-address-sanitizer, 32
--with-app-store-compliance, 35
                                               command line option, 27
--with-assertions, 32
                                           CPPFLAGS, 38, 41
```

command line option, 27	cc, 39
CPython, 88	CCSHARED, 39
current context, 88	CFLAGS, 30, 39, 40
CURSES_CFLAGS	CFLAGS_ALIASING, 39
command line option, 27	CFLAGS_NODIST, 39, 40
CURSES_LIBS	CFLAGSFORSHARED, 39
command line option, 27	COMPILEALL_OPTS, 39
ornamenta and or contract	CONFIGURE_CFLAGS, 39
D	CONFIGURE_CFLAGS_NODIST, 39
- -d	CONFIGURE CPPFLAGS, 38
	CONFIGURE_LDFLAGS, 40
command line option, 6	CONFIGURE_LDFLAGS_NODIST, 40
decorator, 88	CPPFLAGS, 38, 41
descriptor, 89	CXX, 39
dictionary, 89	EXTRA_CFLAGS, 39
dictionary comprehension, 89	LDFLAGS, 38, 40, 41
dictionary view, 89	LDFLAGS_NODIST, 40
disable-gil	LDSHARED, 41
command line option, 26	•
disable-ipv6	LIBS, 41
command line option, 24	LINKCC, 40
disable-test-modules	OPT, 32, 39
command line option, 29	PATH, 10, 20, 44, 46, 5254, 56
docstring, 89	PATHEXT, 46
duck-typing, 89	PROFILE_TASK, 30
_	PURIFY, 40
E	PY_BUILTIN_MODULE_CFLAGS, 40
-E	PY_CFLAGS, 40
command line option, 6	PY_CFLAGS_NODIST, 40
EAFP, 89	PY_CORE_CFLAGS, 40
enable-big-digits	PY_CORE_LDFLAGS, 41
command line option, 24	PY_CPPFLAGS, 38
enable-bolt	PY_LDFLAGS, 41
command line option, 30	PY_LDFLAGS_NODIST, 41
enable-experimental-jit	PY_PYTHON, 57
command line option, 27	PY_STDMODULE_CFLAGS, 40
enable-framework	PYLAUNCHER_ALLOW_INSTALL, 58
	PYLAUNCHER_ALWAYS_INSTALL, 58
command line option, 35enable-loadable-sqlite-extensions	PYLAUNCHER_DEBUG, 58
command line option, 24	PYLAUNCHER_DRYRUN, 58
- · · · · · · · · · · · · · · · · · · ·	PYLAUNCHER_NO_SEARCH_PATH, 56
enable-optimizations	PYTHON_BASIC_REPL, 16
command line option, 29	PYTHON_COLORS, 10, 16
enable-profiling	PYTHON_CPU_COUNT, 9, 15
command line option, 31	PYTHON_FROZEN_MODULES, 9, 16
enable-pystats	PYTHON_GIL, 10, 16, 92
command line option, 25	PYTHON_HISTORY, 16
enable-shared	PYTHON_PERF_JIT_SUPPORT, 9, 15
command line option, 33	PYTHON_PRESITE, 10, 16
enable-universalsdk	PYTHONASYNCIODEBUG, 13
command line option, 34	PYTHONBREAKPOINT, 11
enable-wasm-dynamic-linking	PYTHONCASEOK, 11
command line option, 29	
enable-wasm-pthreads	PYTHONCOERCECLOCALE, 14, 25
command line option, 29	PYTHONDEBUG, 6, 11, 31
environment variable	PYTHONDEVMODE, 9, 15
%APPDATA%, 49	PYTHONDONTWRITEBYTECODE, 6, 12
BASECFLAGS, 39	PYTHONDUMPREFS, 16, 32
BASECPPFLAGS, 38	PYTHONDUMPREFSFILE, 16
BLDSHARED, 41	PYTHONEXECUTABLE, 12

PYTHONFAULTHANDLER, 8, 13	global interpreter lock, 92
PYTHONHASHSEED, 7, 12	Н
PYTHONHOME, 6, 10, 59	
PYTHONINSPECT, 6, 11	-h
PYTHONINTMAXSTRDIGITS, 9, 12	command line option, 5
PYTHONIOENCODING, 12, 14	hash-based pyc, 92
PYTHONLEGACYWINDOWSFSENCODING, 14	hashable, 92
PYTHONLEGACYWINDOWSSTDIO, 12, 14	help
PYTHONMALLOC, 13, 14, 31	command line option, 5
PYTHONMALLOCSTATS, 14	help-all
PYTHONNODEBUGRANGES, 9, 15	command line option, 5
PYTHONOUSERSITE, 7, 12	help-env
PYTHONOPTIMIZE, 6, 11	command line option, 5
PYTHONPATH, 6, 10, 11, 52, 58, 59	help-xoptions
PYTHONPERFSUPPORT, 9, 15	command line option, 5
PYTHONPLATLIBDIR, 11	host
PYTHONPROFILEIMPORTTIME, 9, 13	command line option, 35
PYTHONPYCACHEPREFIX, 9, 12	HOSTRUNNER
PYTHONSAFEPATH, 7, 11	command line option, 36
PYTHONSTARTUP, 6, 11	
PYTHONTRACEMALLOC, 8, 13	ı
PYTHONUNBUFFERED, 7, 11	-I
PYTHONUSERBASE, 12 PYTHONUTF8, 9, 15, 53	command line option, 6
	-i
PYTHONVERBOSE, 7, 11 PYTHONWARNDEFAULTENCODING, 9, 15	command line option, 6
	IDLE, 92
PYTHONWARNINGS, 8, 13	immortal, 92
TEMP, 49	immutable, 92
exec-prefix	import path, 92
command line option, 29 expression, 89	importer, 92
extension module, 90	importing, 92
extension module, 70	interactive, 92
F	interpreted, 93
f-string, 90	interpreter shutdown, 93
file object, 90	iterable, 93
file-like object, 90	iterator, 93
filesystem encoding and error handler, 90	J
finder, 90	
floor division, 90	-J
Fortran contiguous, 88	command line option, 10
free threading, 90	K
free variable, 90	
function, 90	key function, 93
function annotation, 90	keyword argument, 93
Tunecton annotation, 70	I
G	L
	lambda, 94
garbage collection, 91	LBYL, 94
GDBM_CFLAGS	LDFLAGS, 38, 40, 41
command line option, 27	command line option, 27
GDBM_LIBS	LDFLAGS_NODIST, 40
command line option, 28	LIBB2_CFLAGS
generator, 91	command line option, 28
generator expression, 91	LIBB2_LIBS
generator iterator, 91	command line option, 28
generic function, 91	LIBEDIT_CFLAGS
generic type, 91	command line option, 28
GIL, 91	LIBEDIT_LIBS

command line option, 28	Ο
LIBFFI_CFLAGS	-0
command line option, 28	command line option, 6
LIBFFI_LIBS	object, 96
command line option, 28	-00
LIBLZMA_CFLAGS	command line option, 6
command line option, 28	OPT, 32
LIBLZMA_LIBS	optimized scope, 96
command line option, 28	
LIBMPDEC_CFLAGS	Р
command line option, 28	-P
LIBMPDEC_LIBS	command line option, 6
command line option, 28	package, 96
LIBREADLINE_CFLAGS	PANEL_CFLAGS
command line option, 28	command line option, 28
LIBREADLINE_LIBS	PANEL_LIBS
command line option, 28	command line option, 28
command line option, 27	parameter, 96
LIBSQLITE3_CFLAGS	PATH, 10, 20, 44, 46, 5254, 56
command line option, 28	path based finder, 97
LIBSQLITE3_LIBS	path entry, 97
command line option, 28	path entry finder, 97
LIBUUID_CFLAGS	path entry hook, 97
command line option, 28	path-like object, 97
LIBUUID_LIBS	PATHEXT, 46
command line option, 28	PEP, 97
list, 94	PKG_CONFIG
list comprehension, 94	command line option, 27
loader, 94	PKG_CONFIG_LIBDIR
locale encoding, 94	command line option, 27 PKG_CONFIG_PATH
N /	command line option, 27
M	portion, 97
-m	positional argument, 97
command line option, 3	prefix
MACHDEP	command line option, 29
command line option, 27	PROFILE_TASK, 30
magic	provisional API, 97
method, 94	provisional package, 97
magic method, 94	PY_PYTHON, 57
mapping, 94	PYLAUNCHER_ALLOW_INSTALL, 58
meta path finder, 94	PYLAUNCHER_ALWAYS_INSTALL, 58
metaclass, 94	PYLAUNCHER_DEBUG, 58
method, 95	PYLAUNCHER_DRYRUN, 58
magic, 94 special, 99	PYLAUNCHER_NO_SEARCH_PATH, 56
*	Python 3000, 97
method resolution order, 95	Python Enhancement Proposals
module, 95 module spec, 95	PEP 1,97
MRO, 95	PEP 7, 23
mutable, 95	PEP 8, 83
macasic, , , ,	PEP 11, 23, 43, 60
N	PEP 238, 90
named tuple, 95	PEP 278, 100
namespace, 95	PEP 302, 94
namespace package, 95	PEP 338,4
nested scope, 95	PEP 343, 88
new-style class, 96	PEP 362, 86, 96
now begin crabb, >0	PEP 370, 7, 12

PEP 397,54	Q
PEP 411,97	-q
PEP 420, 95, 97	command line option, 7
PEP 443,91	qualified name, 98
PEP 483,91	_
PEP 484, 85, 91, 100	R
PEP 488,6	-R
PEP 492, 86, 88	command line option, 7
PEP 498, 90	reference count, 98
PEP 514,54	regular package, 98
PEP 519,97	REPL, 98
PEP 525, 86	
PEP 526, 85, 100	S
PEP 528,53	-S
PEP 529, 14, 53 PEP 538, 15, 25	command line option, 7
PEP 585, 91	-s
PEP 683, 92	command line option, 7
PEP 703, 48, 70, 90, 92	sequence, 98
PEP 3116, 100	set comprehension, 99
PEP 3155, 98	single dispatch, 99
PYTHON_COLORS, 10	slice, 99
PYTHON_CPU_COUNT, 9	soft deprecated, 99
PYTHON_FROZEN_MODULES, 9	special
PYTHON_GIL, 10, 92	method, 99
PYTHON_PERF_JIT_SUPPORT, 9	special method, 99
PYTHON_PRESITE, 10	statement, 99
PYTHONCOERCECLOCALE, 25	static type checker, 99
PYTHONDEBUG, 6, 31	strong reference, 99
PYTHONDEVMODE, 9	Т
PYTHONDONTWRITEBYTECODE, 6	ı
PYTHONDUMPREFS, 32	TCLTK_CFLAGS
PYTHONFAULTHANDLER, 8	command line option, 2
PYTHONHASHSEED, 7, 12	TCLTK_LIBS
PYTHONHOME, 6, 10, 59	command line option, 2
Pythonic, 97	TEMP, 49
PYTHONINSPECT, 6	text encoding, 99 text file, 99
PYTHONINTMAXSTRDIGITS, 9	triple-quoted string, 99
PYTHONIOENCODING, 14	type, 100
PYTHONLEGACYWINDOWSSTDIO, 12	type alias, 100
PYTHONMALLOC, 14, 31	type hint, 100
PYTHONNODEBUGRANGES, 9	cype nine, 100
PYTHONNOUSERSITE, 7 PYTHONOPTIMIZE, 6	U
PYTHONOPITMIZE, 0 PYTHONPATH, 6, 10, 11, 52, 58, 59	-u
PYTHONPERFSUPPORT, 9	command line option, 7
PYTHONPROFILEIMPORTTIME, 9	universal newlines, 100
PYTHONPYCACHEPREFIX, 9	
PYTHONSAFEPATH, 7	V
PYTHONSTARTUP, 6, 11	-V
PYTHONTRACEMALLOC, 8	command line option, 5
PYTHONUNBUFFERED, 7	-v
PYTHONUTF8, 9, 15, 53	command line option, 7
PYTHONVERBOSE, 7	variable annotation, 100
PYTHONWARNDEFAULTENCODING, 9	version
PYTHONWARNINGS, 8	command line option, 5
	virtual environment, 100
	virtual environment, 100 virtual machine, 101

W	command line option, 25
-W	with-platlibdir
command line option, 7	command line option, 25
with-address-sanitizer	with-pydebug
command line option, 32	command line option, 32
with-app-store-compliance	with-readline
command line option, 35	command line option, 33
with-assertions	with-ssl-default-suites
command line option, 32	command line option, 34
with-build-python	with-strict-overflow
command line option, 36	command line option, 31
with-builtin-hashlib-hashes	with-suffix
command line option, 34	command line option, 24
with-computed-gotos	with-system-expat
command line option, 31	command line option, 33
with-dbmliborder	with-system-libmpdec
command line option, 25	command line option, 33
with-dtrace	with-thread-sanitizer
command line option, 32	command line option, 32
with-emscripten-target	with-trace-refs
command line option, 29	command line option, 32
with-ensurepip	with-tzpath
command line option, 29	command line option, 24
with-framework-name	with-undefined-behavior-sanitizer
command line option, 35	command line option, 32
with-hash-algorithm	with-universal-archs
command line option, 34	command line option, 35
with-libc	with-valgrind
command line option, 33	command line option, 32
with-libm	with-wheel-pkg-dir
command line option, 33	command line option, 25
with-libs	X
command line option, 33	^
with-lto	-X
command line option, 30	command line option, 8
with-memory-sanitizer	-X
command line option, 32	command line option, 8
with-openssl	7
command line option, 33	Z
with-openssl-rpath	Zen of Python, 101
command line option, 34	ZLIB_CFLAGS
without-c-locale-coercion	command line option, 28
command line option, 25	ZLIB_LIBS
without-decimal-contextvar	command line option, 29
command line option, 24	
without-doc-strings	
command line option, 31	
without-freelists	
command line option, 25	
without-mimalloc	
command line option, 31	
without-pymalloc	
command line option, 31	
without-readline	
command line option, 33	
without-static-libpython	
command line option, 33	
with-pkg-config	