Python Setup and Usage

發𝔽 *3.13.2*

Guido van Rossum and the Python development team

3 月 03, 2025

Python Software Foundation
Email: docs@python.org

Contents

這部分的說明文件是關於在不同平台上設定 Python 環境的綜合資訊、直譯器的呼叫，以及讓 Python 更容易使用的一些方法。

CHAPTER 1

命令列與環境

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

CPython 實作細節： Other implementations' command line schemes may differ. See implementations for further resources.

1.1 命令列

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 介面選項

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).

引發一個附帶引數 `command` 的稽核事件 `cpython.run_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.

備F

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
```

引發一個附帶引數 `module-name` 的稽核事件 `cpython.run_module`。

也參考

runpy.run_module()
Equivalent functionality directly available to Python code

PEP 338 -- Executing modules as scripts

在 3.1 版的變更: Supply the package name to run a `__main__` submodule.

在 3.4 版的變更: 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`.

引發一個不附帶引數的稽核事件 `cpython.run_stdin`。

`<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.

引發一個附帶引數 `filename` 的稽核事件 `cpython.run_file`。

也參考

`runpy.run_path()`
Equivalent functionality directly available to Python code

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).

也參考

tut-invoking

在 3.4 版的變更: 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.

在 3.11 版被加入.

`--help-xoptions`

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

在 3.11 版被加入.

`--help-all`

印出完整使用資訊並離開。

在 3.11 版被加入.

`-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]
```

在 3.6 版被加入: `-VV` 選項

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`).

在 3.5 版的變更: 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.

-E

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

另請參F *-P* 和 *-I* (isolated) 選項。

-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 *PYTHONINSPECT*.

-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.

在 3.4 版被加入.

-O

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*.

在 3.5 版的變更: 根據 **PEP 488** 修改 `.pyc` 檔案名稱。

-OO

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

在 3.5 版的變更: 根據 **PEP 488** 修改 `.pyc` 檔案名稱。

-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.

在 3.11 版被加入.

-q

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

在 3.2 版被加入.

-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.

在 3.2.3 版被加入.

在 3.7 版的變更: The option is no longer ignored.

-s

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

另請參F *PYTHONNOUSERSITE*。

也參考

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.

另請參F *PYTHONUNBUFFERED*。

在 3.7 版的變更: 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.

在 3.10 版的變更: The `site` module reports the site-specific paths and `.pth` files being processed.

另請參🄵 *PYTHONVERBOSE*。

-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`.

完整的引數形式🄵:

```
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 *-W* options can be given; when a warning matches more than one option, the action for the last matching option is performed. Invalid *-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*.

  在 3.3 版被加入.

- `-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*.

  在 3.4 版被加入.

- `-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.

  在 3.4 版被加入.

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

  在 3.11 版被加入.

- `-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 multi-threaded application. Typical usage is `python3 -X importtime -c 'import asyncio'`. See also *PYTHONPROFILEIMPORTTIME*.

  在 3.7 版被加入.

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

  在 3.7 版被加入.

- `-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*.

  在 3.7 版被加入.

- `-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*.

  在 3.8 版被加入.

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

  在 3.10 版被加入.

- `-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*.

  在 3.11 版被加入.

- `-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*.

  在 3.11 版被加入.

- `-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.

  在 3.12 版被加入.

- `-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.

在 3.13 版被加入.

- `-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.

  在 3.13 版被加入.

- `-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*.

  在 3.13 版被加入.

- `-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.

  在 3.13 版被加入.

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

在 3.2 版被加入.

在 3.9 版的變更: 移除 `-X showalloccount` 選項。

在 3.10 版的變更: 移除 `-X oldparser` 選項。

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 *PYTHON_COLORS* environment variable can be used. This variable takes precedence over `NO_COLOR`, which in turn takes precedence over `FORCE_COLOR`.

1.1.5 你不該使用的選項

-J

Reserved for use by Jython.

1.2 環境變數

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 `prefix/lib/pythonversion` and `exec_prefix/lib/pythonversion`, where `prefix` and `exec_prefix` are installation-dependent directories, both defaulting to `/usr/local`.

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/python*version* (see *PYTHONHOME* above). It is *always* appended to *PYTHONPATH*.

An additional directory will be inserted in the search path in front of *PYTHONPATH* as described above under 介面選項. The search path can be manipulated from within a Python program as the variable `sys.path`.

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.

在 3.11 版被加入.

PYTHONPLATLIBDIR

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

在 3.9 版被加入.

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 `sys.ps1` and `sys.ps2` and the hook `sys.__interactivehook__` in this file.

引發一個附帶呼叫F動時的檔案名稱F引數的稽核事件 `cpython.run_startup`。

PYTHONOPTIMIZE

If this is set to a non-empty string it is equivalent to specifying the *-O* option. If set to an integer, it is equivalent to specifying *-O* 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 `sys.breakpointhook()` which itself is called by built-in `breakpoint()`. 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 `sys.breakpointhook()` to do nothing but return immediately.

在 3.7 版被加入.

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.

引發一個不附帶引數的稽核事件 `cpython.run_stdin`。

在 3.12.5 版的變更: (also 3.11.10, 3.10.15, 3.9.20, and 3.8.20) Emits audit events.

在 3.13 版的變更: 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.

在 3.8 版被加入.

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.

在 3.2.3 版被加入.

PYTHONINTMAXSTRDIGITS

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

在 3.11 版被加入.

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'`.

在 3.4 版的變更: The `encodingname` part is now optional.

在 3.6 版的變更: On Windows, the encoding specified by this variable is ignored for interactive console buffers unless *PYTHONLEGACYWINDOWSSTDIO* 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`.

也參考

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`.

> **也參考**
>
> **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.

在 3.3 版被加入.

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.

在 3.4 版被加入.

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.

在 3.7 版被加入.

PYTHONASYNCIODEBUG

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

在 3.4 版被加入.

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.

在 3.6 版被加入.

在 3.7 版的變更: 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.

在 3.6 版的變更: 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()`.

適用: Windows.

在 3.6 版被加入: 更多細節請見 **PEP 529**。

`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.

適用: Windows.

在 3.6 版被加入.

`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.

適用: Unix.

在 3.7 版被加入: 更多細節請見 **PEP 538**。

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.

在 3.7 版被加入.

PYTHONUTF8

如果設🄵 `1`，則🄵用 Python UTF-8 Mode。

如果設🄵 `0`，則停用 Python UTF-8 Mode。

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

在 3.7 版被加入.

PYTHONWARNDEFAULTENCODING

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

細節請見 io-encoding-warning。

在 3.10 版被加入.

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.

在 3.11 版被加入.

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.

在 3.12 版被加入.

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.

在 3.13 版被加入.

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.

在 3.13 版被加入.

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.

在 3.13 版被加入.

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*.

在 3.13 版被加入.

PYTHON_BASIC_REPL

If this variable is set to any value, 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*.

在 3.13 版被加入.

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).

在 3.13 版被加入.

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.

在 3.13 版被加入.

1.2.1 除錯模式變數

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.

在 3.11 版被加入.

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.

在 3.13 版被加入.

CHAPTER 2

在 Unix 平臺上使用 Python

2.1 獲得□安裝 Python 的最新版本

2.1.1 在 Linux 上

在大多數 Linux 發行版上會預先安裝 Python，□作□一個套件提供給所有其他使用者。但是發行版提供的套件可能□有你想要使用的某些功能，這時你可以選擇從原始碼編譯最新版本的 Python。

如果最新版的 Python □有預先安裝，□且不在發行版提供的儲存庫 (repository) 中，你可以輕鬆地□自己使用的發行版建立套件。參□以下連結：

也參考

https://www.debian.org/doc/manuals/maint-guide/first.en.html
對於 Debian 用□

https://en.opensuse.org/Portal:Packaging
對於 OpenSuse 用□

https://docs.fedoraproject.org/en-US/package-maintainers/Packaging_Tutorial_GNU_Hello/
對於 Fedora 用□

https://slackbook.org/html/package-management-making-packages.html
對於 Slackware 用□

Installing IDLE

In some cases, IDLE might not be included in your Python installation.

- For Debian and Ubuntu users:

```
sudo apt update
sudo apt install idle
```

- For Fedora, RHEL, and CentOS users:

```
sudo dnf install python3-idle
```

- For SUSE and OpenSUSE users:

```
sudo zypper in python3-idle
```

- For Alpine Linux users:

```
sudo apk add python3-idle
```

2.1.2 在 FreeBSD 和 OpenBSD 上

- FreeBSD 用戶應使用以下命令增加套件：

```
pkg install python3
```

- OpenBSD 用戶應使用以下命令增加套件：

```
pkg_add -r python

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

例如 i386 使用者要獲取 Python 2.5.1 的可用版本：

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

2.2 建置 Python

如果你想自己編譯 CPython，首先要做的是獲取原始碼。你可以下載最新版本的原始碼，也可以直接提取最新的 clone（克隆）。（如果你想要貢獻修補程式碼，也會需要一份 clone。）

建置過程由幾個常用命令組成：

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

特定 Unix 平臺的配置選項和注意事項通常會詳細地記錄在 Python 原始碼樹 (source tree) 根目錄下的 README.rst 檔案中。

警告

`make install` 可以覆蓋或偽裝 `python3` 二進位制檔案。因此，建議使用 `make altinstall` 而不是 `make install`，因為它只安裝 `exec_prefix/bin/pythonversion`。

2.3 與 Python 相關的路徑和檔案

這取決於本地安裝慣例；*prefix* 和 *exec_prefix* 相依於安裝方式，應被直譯來讓 GNU 軟體使用；它們也可能相同。

例如，在大多數 Linux 系統上，兩者的預設值皆是 `/usr`。

檔案/目錄	含意
exec_prefix/bin/python3	直譯器的推薦位置。
prefix/lib/python*version*、*exec_prefix*/lib/python*version*	包含標准模組目錄的推薦位置。
prefix/include/python*version*、*exec_prefix*/include/python*version*	包含開發 Python 擴充套件和嵌入直譯器所需 include 檔案之目錄的推薦位置。

2.4 雜項

要在 Unix 上使用 Python 🄵本，你需要讓他們是可執行的 (executable)，例如用

```
$ chmod +x script
```

🄵在🄵本的頂部放一個合適的 Shebang。以下通常是個好選擇：

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

將在整個 `PATH` 中搜索 Python 直譯器。然而某些 Unix 系統可能🄵有 **env** 命令，因此你可能需要將 `/usr/bin/python3` 寫死 (hardcode) 成直譯器路徑。

要在 Python 🄵本中使用 shell 命令，請見 `subprocess` 模組。

2.5 客🄵化 OpenSSL

1. 要使用你所選擇發行商 (vendor) 的 OpenSSL 配置和系統信任儲存區 (system trust store)，請找到包含 `openssl.cnf` 檔案的目🄵或位於 `/etc` 的符號連結 (symlink)。在大多數發行版上，該檔案會是在 `/etc/ssl` 或者 `/etc/pki/tls` 中。該目🄵亦應包含一個 `cert.pem` 檔案和/或一個 `certs` 目🄵。

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

2. 下載、建置🄵安裝 OpenSSL。請確保你使用 `install_sw` 而不是 `install`。`install_sw` 的目標不會覆蓋 `openssl.cnf`。

```
$ curl -O https://www.openssl.org/source/openssl-VERSION.tar.gz
$ tar xzf openssl-VERSION
$ pushd openssl-VERSION
$ ./config \
    --prefix=/usr/local/custom-openssl \
    --libdir=lib \
    --openssldir=/etc/ssl
$ make -j1 depend
$ make -j8
$ make install_sw
$ popd
```

3. 使用客🄵化 OpenSSL 建置 Python（參見配置 `--with-openssl` 和 `--with-openssl-rpath` 選項）

```
$ 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
```

備🄵

OpenSSL 的修補釋出版 (patch releases) 具有向後相容的 ABI。你不需要重新編譯 Python 來更新 OpenSSL。使用一個新的版本來替代客🄵化 OpenSSL 安裝版就可以了。

CHAPTER 3

配置 Python

3.1 建置需求

建置 CPython 所需的功能與最小版本：

- A C11 compiler. Optional C11 features are not required.
- 在 Windows 上需要 Microsoft Visual Studio 2017 或更新版本。
- Support for IEEE 754 floating-point numbers and floating-point Not-a-Number (NaN).
- thread 的支援。
- 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.

在 3.1 版的變更: Tcl/Tk version 8.3.1 現在是必要的。

在 3.5 版的變更: 在 Windows 上，Visual Studio 2015 或更新版本與 Tcl/Tk version 8.4 現在是必要的。

在 3.6 版的變更: Selected C99 features are now required, like `<stdint.h>` and `static inline` functions.

在 3.7 版的變更: 對執行緒與 OpenSSL 1.0.2 的支援現在是必要的。

在 3.10 版的變更: OpenSSL 1.1.1 現在是必要的。需要 SQLite 3.7.15。

在 3.11 版的變更: 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.

在 3.13 版的變更: 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 $\boxed{F}$生的檔案

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 設定🄵本

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 設定選項

使用以下命令列出所有 `configure` 🄵本選項：

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

請見 Python 原始碼發行版中的 `Misc/SpecialBuilds.txt`。

3.3.1 一般選項

`--enable-loadable-sqlite-extensions`

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

請見 `sqlite3` 模組的 `sqlite3.Connection.enable_load_extension()` 方法。

在 3.6 版被加入.

`--disable-ipv6`

停用 IPv6 支援（如果支援的話預設是🄵用的），請見 `socket` 模組。

`--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.

將 `PYLONG_BITS_IN_DIGIT` 定義🄵 `15` 或 `30`。

參🄵 `sys.int_info.bits_per_digit`。

`--with-suffix``=SUFFIX`

將 Python 執行檔的後綴設定🄵 *SUFFIX*。

在 Windows 和 macOS 上預設的後綴是 `.exe`（`python.exe` 執行檔）、在 Emscripten node 上🄵 `.js`、在 Emscripten 🄵覽器🄵 `.html`、在 WASI 上🄵 `.wasm`，以及在其他平台🄵空字串（`python` 執行檔）。

在 3.11 版的變更: 在 WASM 平台上預設的後綴是 `.js`、`.html` 或 `.wasm` 中的一個。

`--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.

預設值：`/usr/share/zoneinfo:/usr/lib/zoneinfo:/usr/share/lib/zoneinfo:/etc/zoneinfo`。

請見 os.pathsep 路徑分隔符號。

在 3.9 版被加入.

--without-decimal-contextvar

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

請見 decimal.HAVE_CONTEXTVAR 與 contextvars 模組。

在 3.9 版被加入.

--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).

不要定義 PY_COERCE_C_LOCALE 巨集。

請見 *PYTHONCOERCECLOCALE* 與 **PEP 538**。

--without-freelists

Disable all freelists except the empty tuple singleton.

在 3.11 版被加入.

--with-platlibdir=DIRNAME

Python 函式庫目F名稱（預設F lib）。

Fedora 和 SuSE 在 64 位元平台上使用 lib64。

參F sys.platlibdir。

在 3.9 版被加入.

--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.

在 3.10 版被加入.

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

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

- check（預設）：**pkg-config** 是可選的
- yes：**pkg-config** 是必要的
- no：即使存在也不使用 **pkg-config** 來配置

在 3.11 版被加入.

--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.

效果：

- 新增*-X pystats* 命令列選項。
- 新增 `PYTHONSTATS` 環境變數。
- 定義 `Py_STATS` 巨集。
- 新增函式到 `sys` 模組。
  - `sys._stats_on()`： 🄵用統計資料收集。
  - `sys._stats_off()`： 關閉統計資料收集。
  - `sys._stats_clear()`： 清除統計資料。
  - `sys._stats_dump()`： 將統計資料轉儲到檔案，🄵清除統計資料。

統計資料將被轉儲到 `/tmp/py_stats/` (Unix) 或 `C:\temp\py_stats\` (Windows) 中的任意（可能是唯一的）檔案。如果該目🄵不存在，結果將被列印到 stderr。

使用 `Tools/scripts/summarize_stats.py` 來讀取統計資料。

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.
- 物件：
  - incref and decref;
  - interpreter incref and decref;
  - allocations: all, 512 bytes, 4 kiB, big;
  - free;
  - to/from free lists;
  - dictionary materialized/dematerialized;
  - type cache;
  - 最佳化嘗試；
  - optimization traces created/executed;
  - uops executed.
- Garbage collector:
  - Garbage collections;
  - Objects visited;
  - Objects collected.

在 3.11 版被加入.

--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.

在 3.13 版被加入.

--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, `--enable-experimental-jit` is a shorthand for `--enable-experimental-jit=yes`.

備F

When building CPython with JIT enabled, ensure that your system has Python 3.11 or later installed.

在 3.13 版被加入.

PKG_CONFIG

Path to `pkg-config` utility.

PKG_CONFIG_LIBDIR

PKG_CONFIG_PATH

`pkg-config` 選項。

3.3.2 C 編譯器選項。

CC

C 編譯器指令。

CFLAGS

C 編譯器旗標。

CPP

C 預處理器指令。

CPPFLAGS

C 預處理器旗標，例如：`-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

在 3.11 版被加入.

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`.

> **備F**
>
> 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 選項

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

Set build flavor for `wasm32-emscripten`.

- `browser` (default): preload minimal stdlib, default MEMFS.
- `node`： 對 NODERAWFS 和 pthread 支援。

在 3.11 版被加入.

--enable-wasm-dynamic-linking

Turn on dynamic linking support for WASM.

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

在 3.11 版被加入.

--enable-wasm-pthreads

Turn on pthreads support for WASM.

在 3.11 版被加入.

3.3.6 安裝選項

--prefix=PREFIX

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

這個值可以在 runtime 使用 `sys.prefix` 取得。

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`*.

這個值可以在 runtime 使用 `sys.exec_prefix` 取得

--disable-test-modules

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

在 3.10 版被加入.

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

選擇在 Python 安裝時執行的 `ensurepip` 命令：

- `upgrade`（預設）：執行 `python -m ensurepip --altinstall --upgrade` 命令。
- `install`：執行 `python -m ensurepip --altinstall` 命令；
- `no`：不要執行 ensurepip；

在 3.6 版被加入.

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 `llvm-profdata` 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.

備F

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*.

在 3.6 版被加入.

在 3.10 版的變更: 在 GCC 上使用 `-fno-semantic-interposition`。

PROFILE_TASK

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

預設值：`-m test --pgo --timeout=$(TESTTIMEOUT)`。

在 3.8 版被加入.

在 3.13 版的變更: 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 `llvm-ar` for LTO (`ar` on macOS), as well as an LTO-aware linker (`ld.gold` or `lld`).

在 3.6 版被加入.

在 3.11 版被加入: To use ThinLTO feature, use `--with-lto=thin` on Clang.

在 3.12 版的變更: 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 `llvm-bolt` and `merge-fdata` 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.

在 3.12 版被加入.

`BOLT_APPLY_FLAGS`

Arguments to `llvm-bolt` when creating a BOLT optimized binary.

在 3.12 版被加入.

`BOLT_INSTRUMENT_FLAGS`

Arguments to `llvm-bolt` when instrumenting binaries.

在 3.12 版被加入.

`--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).

另請參F *`PYTHONMALLOC`* 環境變數。

`--without-pymalloc`

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

另請參F *`PYTHONMALLOC`* 環境變數。

`--without-doc-strings`

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

不要定義 `WITH_DOC_STRINGS` 巨集。

請見 `PyDoc_STRVAR()` 巨集。

`--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.
- 新增 `d` 到 `sys.abiflags`。
- 新增 `sys.gettotalrefcount()` 函式。
- 新增*`-X showrefcount`* 命令列選項。
- 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.
- 定義 `Py_DEBUG` 和 `Py_REF_DEBUG` 巨集。

- 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.

在 3.8 版的變更: 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).

效果:

- 定義 `Py_TRACE_REFS` 巨集。
- 新增 `sys.getobjects()` 函式。
- 新增 *PYTHONDUMPREFS* 環境變數。

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

Statically allocated objects are not traced.

在 3.8 版被加入.

在 3.13 版的變更: 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.

在 3.6 版被加入.

--with-valgrind

F用 Valgrind 支援（預設不F用）。

--with-dtrace

F用 DTrace 支援（預設不F用）。

See Instrumenting CPython with DTrace and SystemTap.

在 3.6 版被加入.

--with-address-sanitizer

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

在 3.6 版被加入.

--with-memory-sanitizer

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

在 3.6 版被加入.

--with-undefined-behavior-sanitizer

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

在 3.6 版被加入.

--with-thread-sanitizer

□用 ThreadSanitizer 資料競□偵測器 `tsan`（預設不□用）。

在 3.13 版被加入.

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).

在 3.10 版被加入.

3.3.11 函式庫選項

--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).

在 3.3 版被加入.

在 3.13 版的變更: 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.

也參考

LIBMPDEC_CFLAGS 和*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.

在 3.10 版被加入.

--without-readline

不要建置 `readline` 模組（預設會建置）。

不要定義 `HAVE_LIBREADLINE` 巨集。

在 3.10 版被加入.

--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

OpenSSL 目🄵的根目🄵。

在 3.7 版被加入.

--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-openssl* and `pkg-config`;
- *DIR*: set an explicit rpath.

在 3.10 版被加入.

3.3.12 安全性選項

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

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

- `siphash13`（預設）;
- `siphash24`;
- `fnv`。

在 3.4 版被加入.

在 3.11 版被加入: `siphash13` is added and it is the new default.

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

🄵建雜🄵模組：

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

在 3.9 版被加入.

--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

請見 `ssl` 模組。

在 3.7 版被加入.

在 3.10 版的變更: The settings `python` and *STRING* also set TLS 1.2 as minimum protocol version.

3.3.13 macOS 選項

參F 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.

選項:

- `universal2` (x86-64 and arm64);
- `32-bit` (PPC and i386);
- `64-bit` (PPC64 and x86-64);
- `3-way` (i386, PPC and x86-64);
- `intel` (i386 and x86-64);
- `intel-32` (i386);
- `intel-64` (x86-64);
- `all` (PPC, i386, PPC64 and x86-64).

Note that values for this configuration item are *not* the same as the identifiers used for universal binary wheels on macOS. See the Python Packaging User Guide for details on the packaging platform compatibility tags used on macOS

`--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.

在 3.13 版被加入.

3.3.14 iOS 選項

参🄵 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

指定框架的名稱（預設值：`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

在 3.11 版被加入.

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.

在 3.11 版被加入.

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 建置系統

3.4.1 建置系統的主要檔案

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

3.4.2 主要建置步驟

- C files (`.c`) are built as object files (`.o`).
- A static `libpython` library (`.a`) is created from objects files.
- `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 主要 Makefile 目標

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

移除建置的檔案。

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.[1]

make install

建置 `all` 目標F安裝 Python。

[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.

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*.

3.4.4 C 擴充模組

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:

```
>>> import _asyncio
>>> _asyncio
<module '_asyncio' from '/usr/lib64/python3.9/lib-dynload/_asyncio.cpython-39-x86_64-linux-
↪gnu.so'>
>>> _asyncio.__file__
'/usr/lib64/python3.9/lib-dynload/_asyncio.cpython-39-x86_64-linux-gnu.so'
```

`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:

- 如果定義了 `Py_BUILD_CORE_MODULE`，則使用 `Py_EXPORTED_SYMBOL`
- 否則使用 `Py_IMPORTED_SYMBOL`。

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 預處理器旗標

CONFIGURE_CPPFLAGS

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

在 3.6 版被加入.

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

在 3.4 版被加入.

PY_CPPFLAGS

Extra preprocessor flags added for building the interpreter object files.

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

在 3.2 版被加入.

3.5.2 編譯器旗標

CC

C 編譯器指令。

範例：`gcc -pthread`。

CXX

C++ 編譯器指令。

範例：`g++ -pthread`。

CFLAGS

C 編譯器旗標。

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.

在 3.5 版被加入.

COMPILEALL_OPTS

Options passed to the `compileall` command line when building PYC files in `make install`. Default: `-j0`.

在 3.12 版被加入.

EXTRA_CFLAGS

額外的 C 編譯器旗標。

CONFIGURE_CFLAGS

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

在 3.2 版被加入.

CONFIGURE_CFLAGS_NODIST

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

在 3.5 版被加入.

BASECFLAGS

基本編譯器旗標。

OPT

最佳化旗標。

CFLAGS_ALIASING

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

在 3.7 版被加入.

CCSHARED

Compiler flags used to build a shared library.

例如F `-fPIC` 被使用於 Linux 與 BSD 上。

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`.

在 3.5 版被加入.

PY_STDMODULE_CFLAGS

C flags used for building the interpreter object files.

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

在 3.7 版被加入.

PY_CORE_CFLAGS

Default: `$(PY_STDMODULE_CFLAGS) -DPy_BUILD_CORE`.

在 3.2 版被加入.

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`.

在 3.8 版被加入.

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.

在 3.2 版被加入.

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.

在 3.8 版被加入.

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.

範例：`-lrt`。

LDSHARED

Command to build a shared library.

預設值：`@LDSHARED@ $(PY_LDFLAGS)`。

BLDSHARED

Command to build `libpython` shared library.

預設值：`@BLDSHARED@ $(PY_CORE_LDFLAGS)`。

PY_LDFLAGS

預設值：`$(CONFIGURE_LDFLAGS) $(LDFLAGS)`。

PY_LDFLAGS_NODIST

預設值：`$(CONFIGURE_LDFLAGS_NODIST) $(LDFLAGS_NODIST)`。

在 3.8 版被加入.

PY_CORE_LDFLAGS

Linker flags used for building the interpreter object files.

在 3.8 版被加入.

CHAPTER 4

在 Windows 上使用 Python

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.

nuget.org 套件 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.

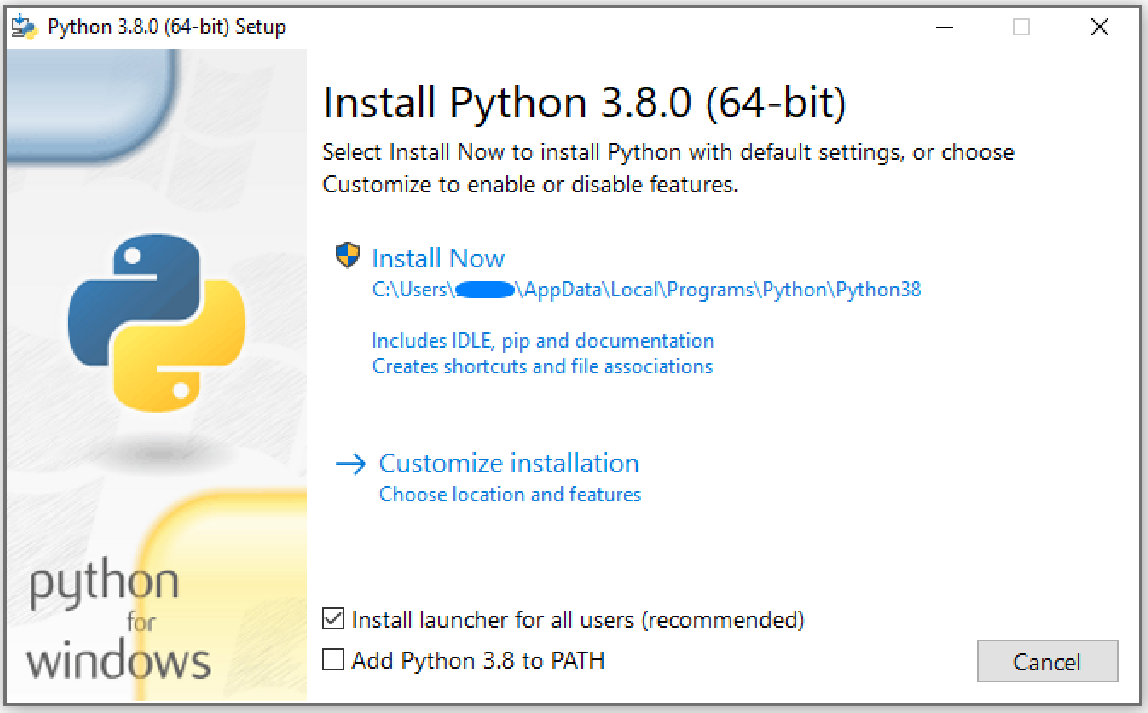
The embeddable package is a minimal package of Python suitable for embedding into a larger application.

4.1 The full installer

4.1.1 安裝步驟

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 當安裝時不下載 for other ways to avoid downloading during installation.

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



Install Python 3.8.0 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\[redacted]\AppData\Local\Programs\Python\Python38

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☐ Add Python 3.8 to PATH

如果你選擇「馬上安裝」:

- 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
- 如果選擇，安裝目🄵將被加入到你的 `PATH`
- 安裝捷徑將只能被目前使用者所看見

選擇「客🄵化安裝」將允許你選擇所需的項目進行安裝，安裝位置與其他選擇或安裝後的所需進行的動作。你將需要使用此選項「除錯特徵」或「二進位方式」進行安裝。

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`
- 捷徑將被所有使用者所見

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 `LongPathsEnabled` to `1` in the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem`.

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.

在 3.6 版的變更: Support for long paths was enabled in Python.

4.1.3 安裝排除使用者介面

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:

名徵	描述
/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.

名稱	描述	預設
InstallAllUsers	Perform a system-wide installation.	0
TargetDir	安裝目🄵	Selected based on InstallAllUsers
DefaultAllUsersTargetDir	The default installation directory for all-user installs	`%ProgramFiles%\Python X.Y` 或 `%ProgramFiles(x86)%\Python X.Y`
DefaultJustForMeTargetDir	預設安裝目🄵給只有給我安裝方式	`%LocalAppData%\Programs\Python\PythonXY` 或 `%LocalAppData%\Programs\Python\PythonXY-32` 或 `%LocalAppData%\Programs\Python\PythonXY-64`
DefaultCustomTargetDir	The default custom install directory displayed in the UI	(empty)
AssociateFiles	當執行程序也被安裝時創造檔案關聯	1
CompileAll	編譯所有 `.py` 檔案🄵 `.pyc`。	0
PrependPath	Prepend install and Scripts directories to `PATH` and add `.PY` to `PATHEXT`	0
AppendPath	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
Include_doc	安裝 Python 文件	1
Include_debu	Install debug binaries	0
Include_dev	Install developer headers and libraries. Omitting this may lead to an unusable installation.	1
Include_exe	Install `python.exe` and related files. Omitting this may lead to an unusable installation.	1
Include_launc	安裝*Python Launcher for Windows*。	1
InstallLauncherAllUsers	Installs the launcher for all users. Also requires `Include_launcher` to be set to 1	1
Include_lib	Install standard library and extension modules. Omitting this may lead to an unusable installation.	1
Include_pip	Install bundled pip and setuptools	1
Include_symb	Install debugging symbols (`*.pdb`)	0
Include_tcltk	Install Tcl/Tk support and IDLE	1
Include_test	Install standard library test suite	1
Include_tools	Install utility scripts	1
LauncherO	Only installs the launcher. This will override most other options.	0
SimpleInstall	Disable most install UI	0

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 當安裝時不下載

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.

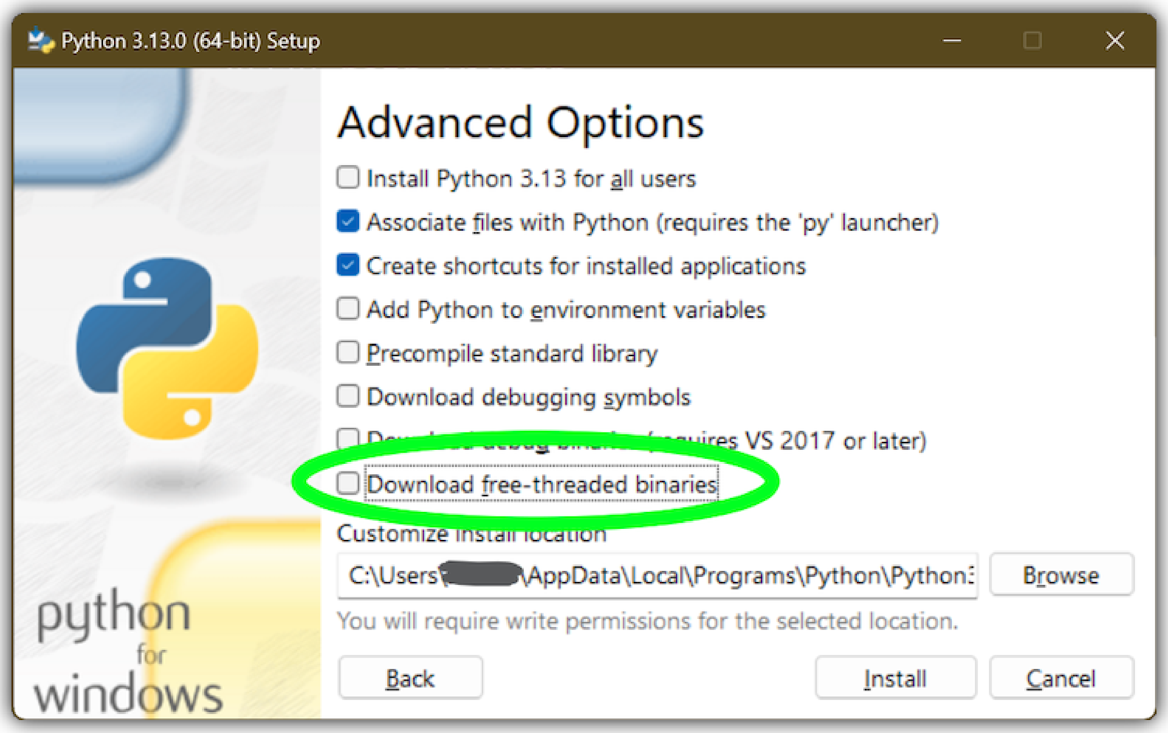
”Uninstall” will remove Python entirely, with the exception of the *Python Launcher for Windows*, which has its own entry in Programs and Features.

4.1.6 Installing Free-threaded Binaries

在 3.13 版被加入: (實驗性的)

> **備F**
>
> 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.



Python 3.13.0 (64-bit) Setup

Advanced Options

- [ ] Install Python 3.13 for all users
- [✓] Associate files with Python (requires the 'py' launcher)
- [✓] Create shortcuts for installed applications
- [ ] Add Python to environment variables
- [ ] Precompile standard library
- [ ] Download debugging symbols
- [ ] Download debug binaries (requires VS 2017 or later)
- [ ] Download free-threaded binaries

Customize install location

C:\Users[redacted]\AppData\Local\Programs\Python\Python3

You will require write permissions for the selected location.

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 當安裝時不下載 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

在 3.7.2 版被加入.

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.

警告

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. For 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 keys.
- 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 nuget.org 套件

在 3.5.2 版被加入.

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

在 3.13 版被加入: (實驗性的)

備🄵

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

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

在 3.5 版被加入.

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`, `python37.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.

備F

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 嵌入 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 設定 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).

備F

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.

也參考

https://learn.microsoft.com/windows/win32/procthread/environment-variables
Windows 上的環境變數概要

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

在 3.5 版的變更.

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 命令列 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 模式

在 3.7 版被加入.

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.

備□

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).

4.8 Python Launcher for Windows

在 3.3 版被加入.

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 per-user 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 開始

From the command-line

在 3.6 版的變更.

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:

```
py
```

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.
```

指令:

```
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:` format.

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/

# 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).

F擬環境 (Virtual environment)

在 3.5 版被加入.

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 #! python3.7 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.

從檔案關聯

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.

在 3.7 版被加入: 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`).

在 3.11 版的變更: 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.

在 3.13 版的變更: 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 `[commands]` 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.

範例：

- 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.

舉例來🄵：

- 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 安裝隨選

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.

名徵	Value	描述
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 completes.
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 找尋模組

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 環境變數, 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 *`PYTHONHOME`* is set, it is assumed as "Python Home". Otherwise, the path of the main Python executable is used to locate a "landmark file" (either `Lib\os.py` or `pythonXY.zip`) to deduce the "Python Home". If a Python home is found, the relevant sub-directories added to `sys.path` (`Lib`, `plat-win`, etc) 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.

最終這所有的結果🄵:

- 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.

在 3.6 版的變更: Add `._pth` file support and removes `applocal` option from `pyvenv.cfg`.

在 3.6 版的變更: Add `python`*`XX`*`.zip` as a potential landmark when directly adjacent to the executable.

在 3.6 版之後被🄵用: 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 額外的模組

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 呼叫
- 登🄵檔（Registry）
- 事件日🄵（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.

也參考

Win32 How Do I...?
    由 Tim Golden 所著

Python and COM
    由 David 與 Paul Boddie 所著

4.10.2 cx_Freeze

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

4.11 編譯 Python 在 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 其他平台

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.

CHAPTER 5

於 macOS 使用 Python

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.

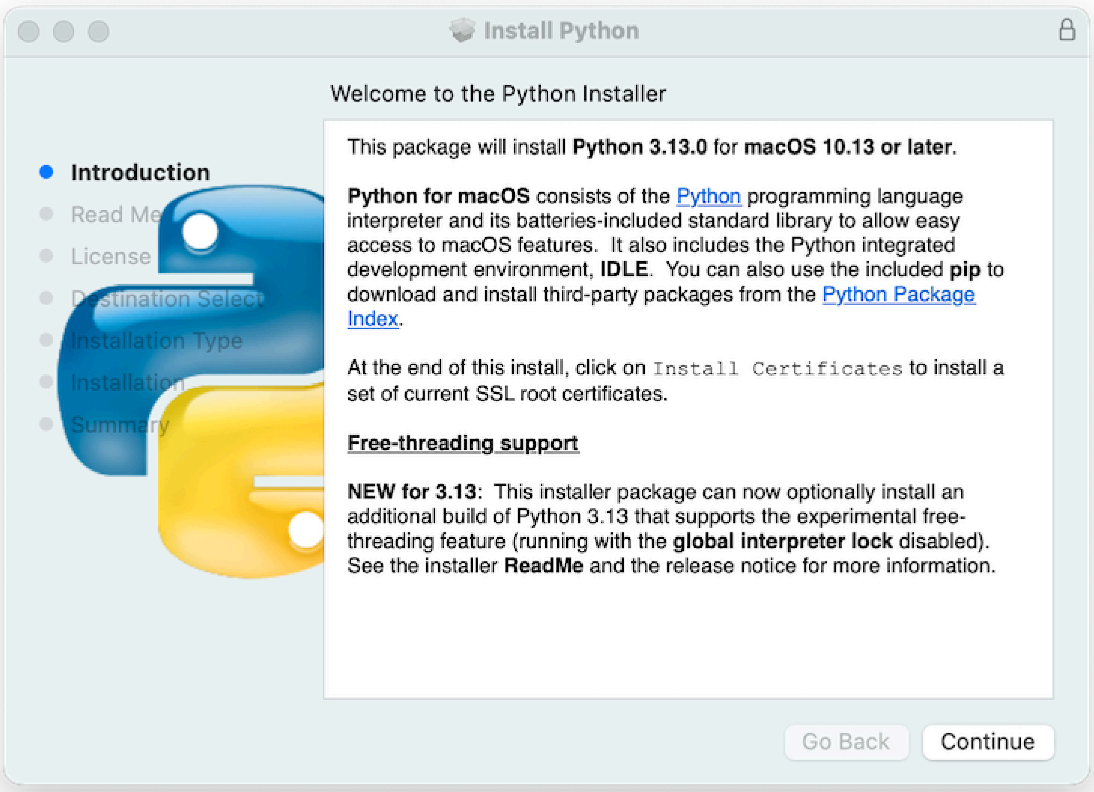
5.1 Using Python for macOS from `python.org`

5.1.1 安裝步驟

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.



Welcome to the Python Installer

This package will install **Python 3.13.0** for **macOS 10.13 or later**.

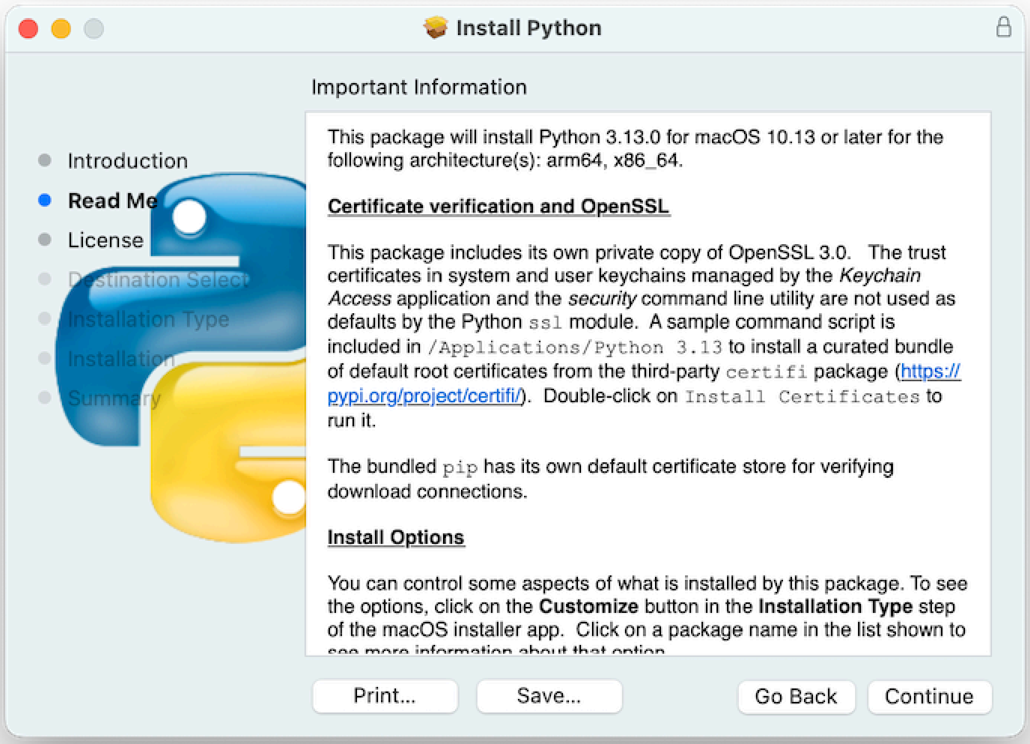
Python for macOS consists of the Python programming language interpreter and its batteries-included standard library to allow easy access to macOS features. It also includes the Python integrated development environment, **IDLE**. You can also use the included **pip** to download and install third-party packages from the Python Package Index.

At the end of this install, click on `Install Certificates` to install a set of current SSL root certificates.

Free-threading support

NEW for 3.13: This installer package can now optionally install an additional build of Python 3.13 that supports the experimental free-threading feature (running with the **global interpreter lock** disabled). See the installer **ReadMe** and the release notice for more information.

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.



Important Information

This package will install Python 3.13.0 for macOS 10.13 or later for the following architecture(s): arm64, x86_64.

Certificate verification and OpenSSL

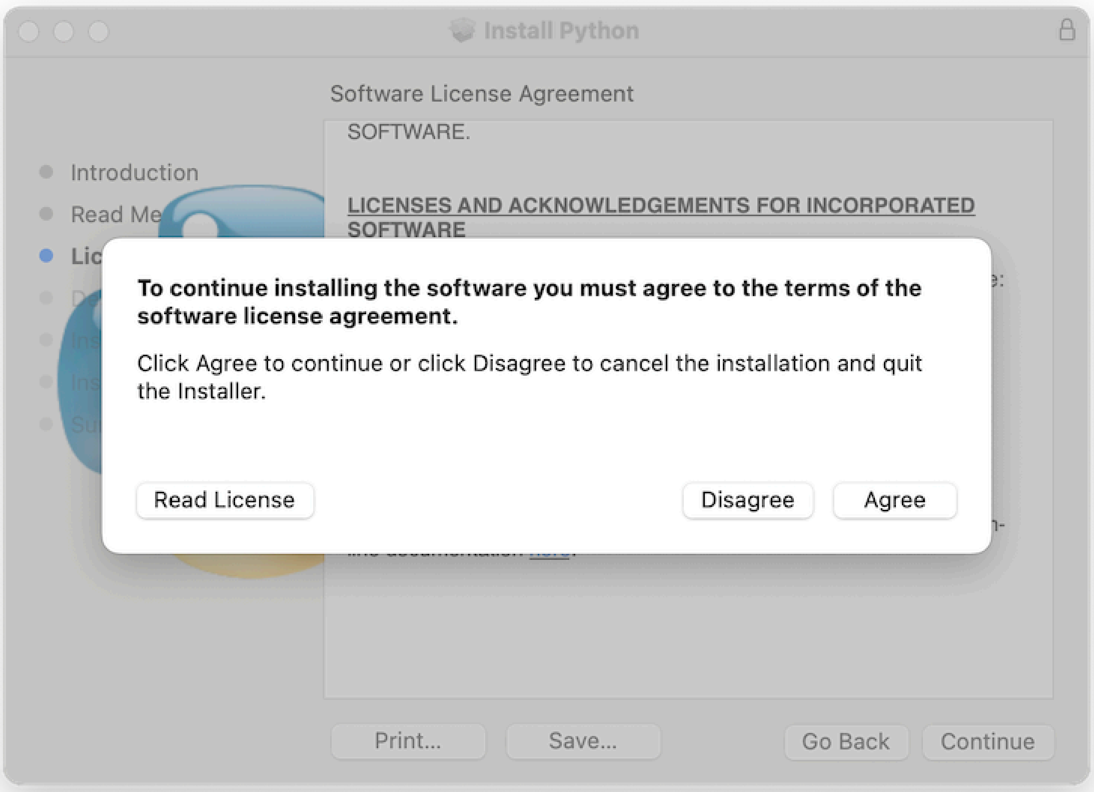
This package includes its own private copy of OpenSSL 3.0. The trust certificates in system and user keychains managed by the *Keychain Access* application and the *security* command line utility are not used as defaults by the Python `ssl` module. A sample command script is included in `/Applications/Python 3.13` to install a curated bundle of default root certificates from the third-party `certifi` package (https://pypi.org/project/certifi/). Double-click on `Install Certificates` to run it.

The bundled `pip` has its own default certificate store for verifying download connections.

Install Options

You can control some aspects of what is installed by this package. To see the options, click on the **Customize** button in the **Installation Type** step of the macOS installer app. Click on a package name in the list shown to

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.


Install Python

Software License Agreement

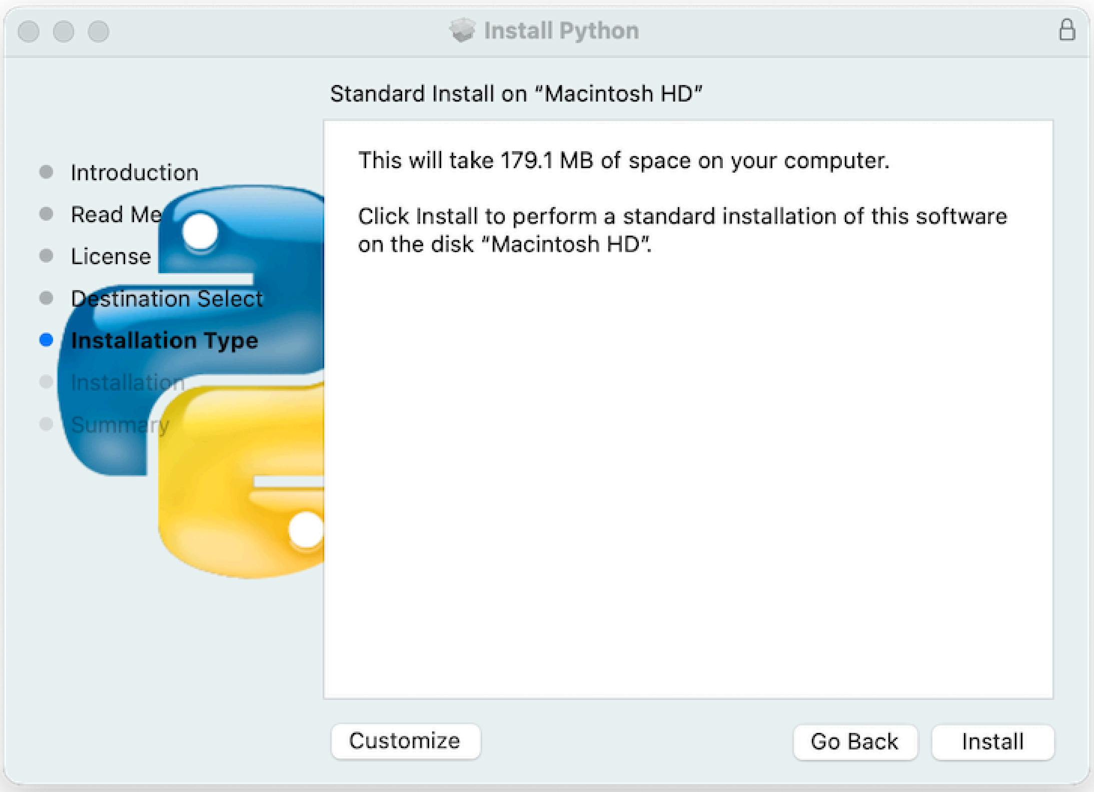
SOFTWARE.

LICENSES AND ACKNOWLEDGEMENTS FOR INCORPORATED SOFTWARE

To continue installing the software you must agree to the terms of the software license agreement.

Click Agree to continue or click Disagree to cancel the installation and quit the Installer.

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.

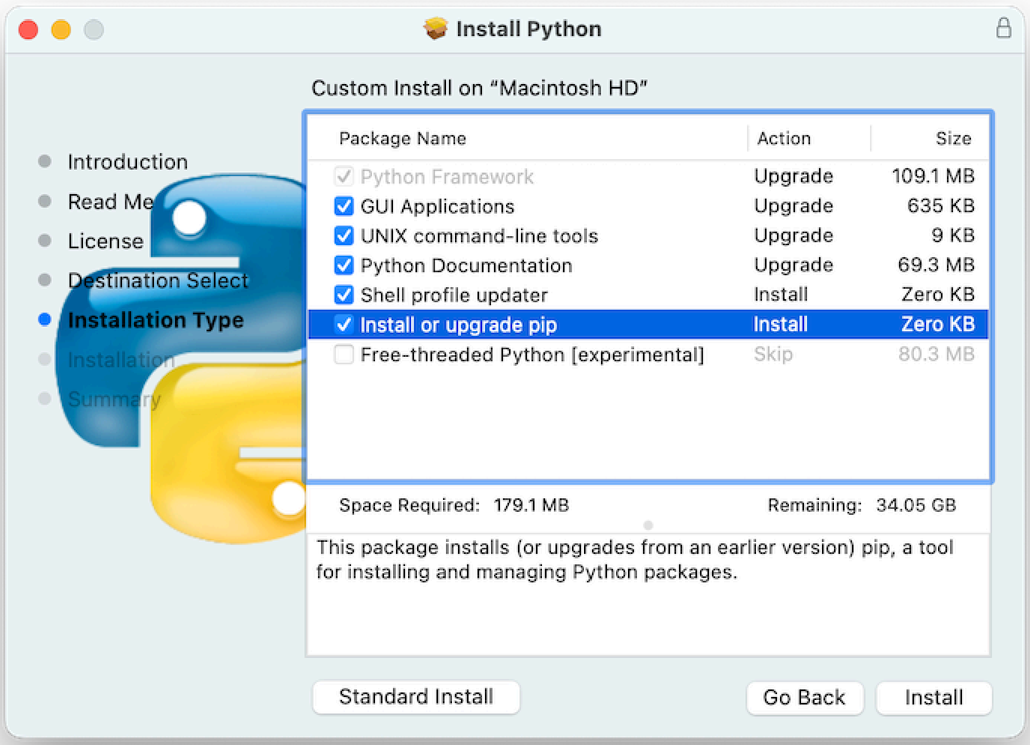


Standard Install on "Macintosh HD"

This will take 179.1 MB of space on your computer.

Click Install to perform a standard installation of this software on the disk "Macintosh HD".

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*.



Custom Install on "Macintosh HD"

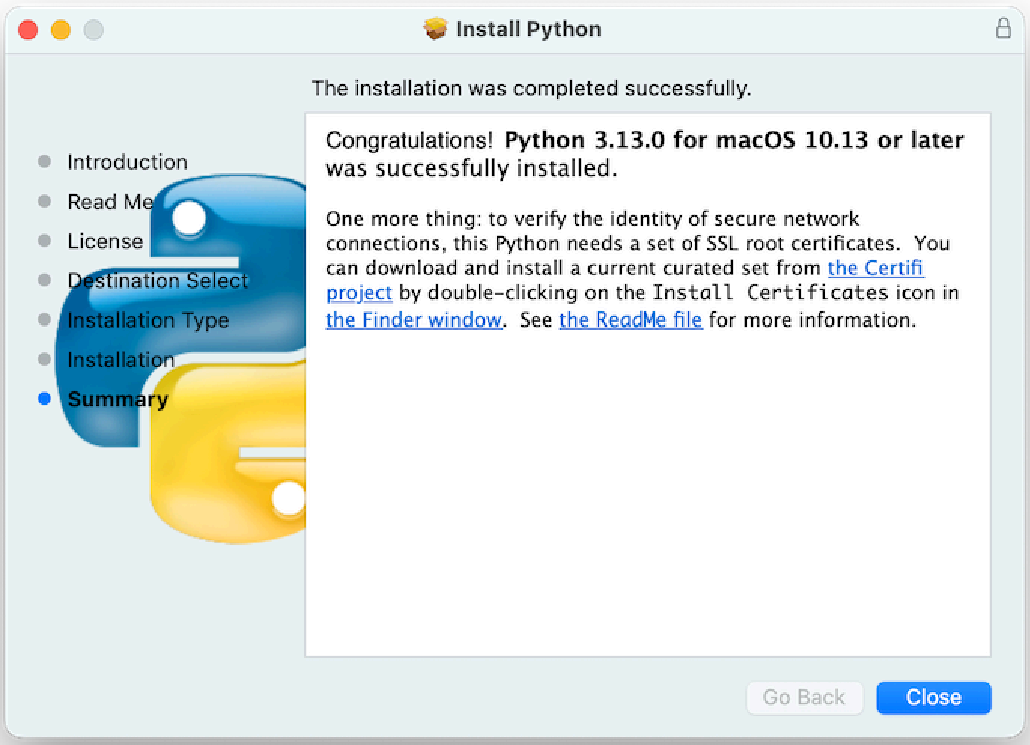
Package Name	Action	Size
✓ Python Framework	Upgrade	109.1 MB
✓ GUI Applications	Upgrade	635 KB
✓ UNIX command-line tools	Upgrade	9 KB
✓ Python Documentation	Upgrade	69.3 MB
✓ Shell profile updater	Install	Zero KB
✓ Install or upgrade pip	Install	Zero KB
Free-threaded Python [experimental]	Skip	80.3 MB

Space Required: 179.1 MB Remaining: 34.05 GB

This package installs (or upgrades from an earlier version) pip, a tool for installing and managing Python packages.

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.

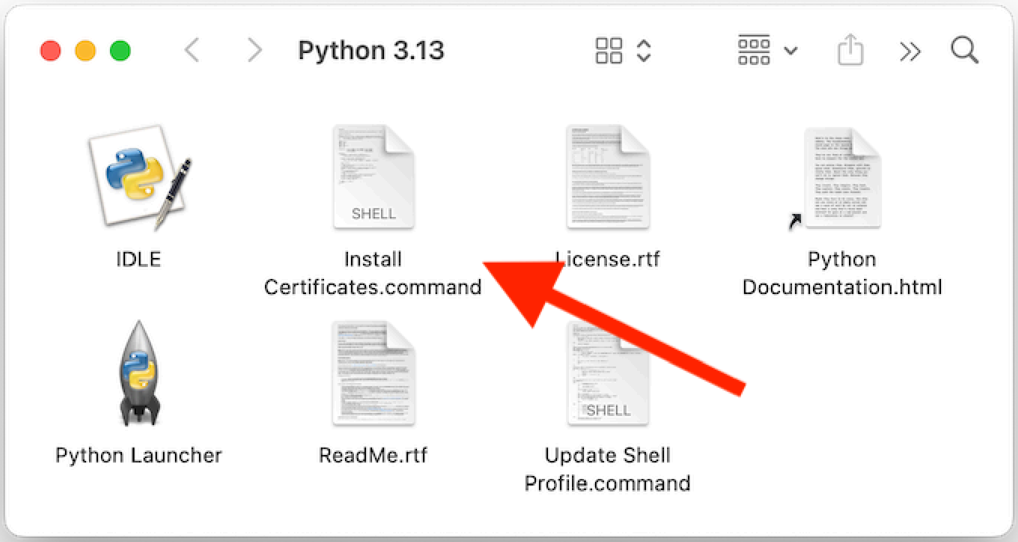


The installation was completed successfully.

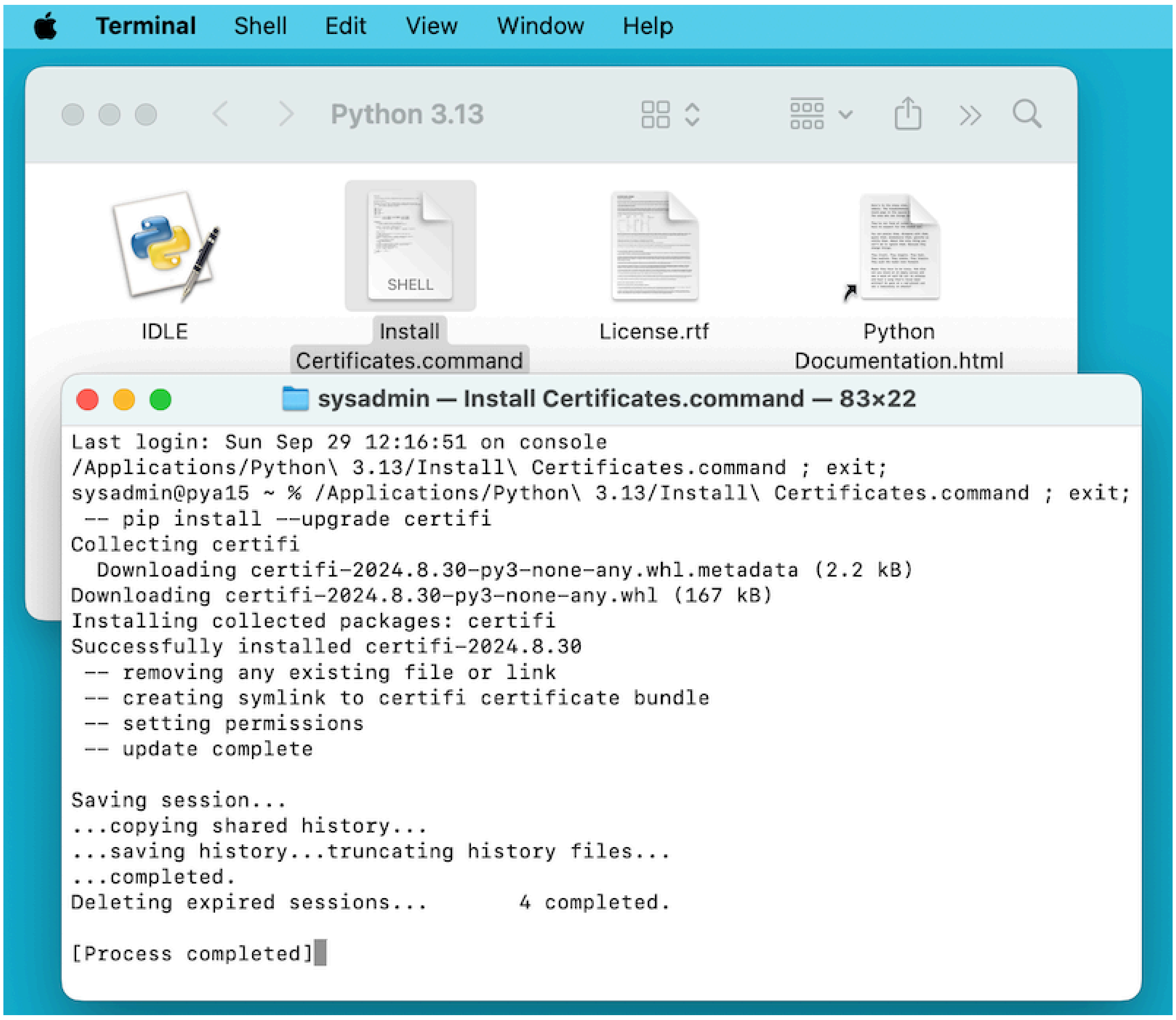
Congratulations! **Python 3.13.0 for macOS 10.13 or later** was successfully installed.

One more thing: to verify the identity of secure network connections, this Python needs a set of SSL root certificates. You can download and install a current curated set from the Certifi project by double-clicking on the `Install Certificates` icon in the Finder window. See the ReadMe file for more information.

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


Python 3.13
IDLE
SHELL
Install Certificates.command
License.rtf
Python Documentation.html
Python Launcher
ReadMe.rtf
SHELL
Update Shell Profile.command

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


Terminal
Shell
Edit
View
Window
Help
Python 3.13
SHELL
IDLE
Install Certificates.command
License.rtf
Python Documentation.html
sysadmin — Install Certificates.command — 83×22

```
Last login: Sun Sep 29 12:16:51 on console
/Applications/Python\ 3.13/Install\ Certificates.command ; exit;
sysadmin@pya15 ~ % /Applications/Python\ 3.13/Install\ Certificates.command ; exit;
 -- pip install --upgrade certifi
Collecting certifi
  Downloading certifi-2024.8.30-py3-none-any.whl.metadata (2.2 kB)
Downloading certifi-2024.8.30-py3-none-any.whl (167 kB)
Installing collected packages: certifi
Successfully installed certifi-2024.8.30
 -- removing any existing file or link
 -- creating symlink to certifi certificate bundle
 -- setting permissions
 -- update complete

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.
Deleting expired sessions...      4 completed.

[Process completed]
```

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

預設安裝會包含：

- 會有一個 `Python 3.13` 資料夾在你的 `Applications` 資料夾中。在這🄵你可以找到 **IDLE**，它是作🄵官方 Python 發行版標准組成的開發環境；以及 **Python Launcher**，它負責處理在 Finder 中雙擊 Python 🄵本的操作。
- `/Library/Frameworks/Python.framework` 框架，包括 Python 可執行檔案 (executable) 和函式庫 (library)。安裝程式將此位置新增到 shell 路徑。要解除安裝 Python，你可以移除這三個東西。Python 可執行檔案的符號連結 (symlink) 則放在 `/usr/local/bin/` 中。

備🄵

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 如何執行 Python 🄵本

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 命令列與環境). 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 *編輯器與 IDE* 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
```

從 Finder 執行你的🄵本時，你可以：

- 把它拖曳到 **Python Launcher**
- 透過 Finder Info 視窗選擇 **Python Launcher** 作🄵開🄵🄵本（或任何 `.py` 🄵本）的預設應用程式，然後雙擊🄵本。**Python Launcher** 有各種用來控制🄵本🄵動方式的選項。拖曳選項可以讓你一次更改多個選項，或使用其 `Preferences` 選單以全域性地更改🄵容。

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.

Homebrew
    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 安裝額外的 Python 套件

更多資訊請見 Python Packaging User Guide。

5.4 開發 GUI 程式

於 Mac 上使用 Python 來建立 GUI 應用程式有許多選項。

標準的 Python GUI 工具套件是基於跨平臺 Tk 工具套件 (https://www.tcl.tk) 的 `tkinter`。macOS 原生版本的 Tk 有包含於安裝程式中。

PyObjC 是一個 Apple Objective-C/Cocoa 框架的 Python 🄵結 (binding)。有關 PyObjC 的資訊請見 pyobjc。

有許多替代 macOS GUI 工具套件可用，包含：

- PySide：Qt GUI 工具包的官方 Python □結。
- PyQt：Qt 的替代 Python □結。
- Kivy：一個支援桌面和行動平臺的跨平臺 GUI 工具包。
- Toga：BeeWare 專案的一部分；支援桌面、行動、網頁和控制台應用程式。
- wxPython：一個支援桌面作業系統的跨平臺工具包。

5.5 進階主題

5.5.1 Installing Free-threaded Binaries

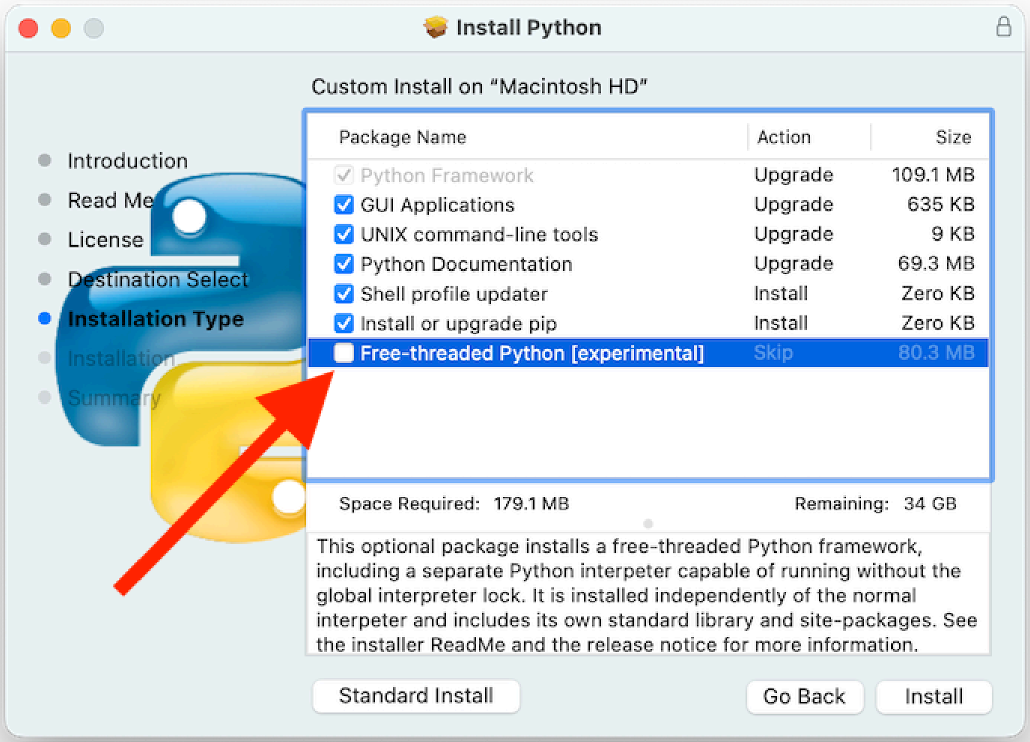
在 3.13 版被加入：（實驗性的）

備□

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.



Custom Install on "Macintosh HD"

	Package Name	Action	Size
✓	Python Framework	Upgrade	109.1 MB
✓	GUI Applications	Upgrade	635 KB
✓	UNIX command-line tools	Upgrade	9 KB
✓	Python Documentation	Upgrade	69.3 MB
✓	Shell profile updater	Install	Zero KB
✓	Install or upgrade pip	Install	Zero KB
☐	Free-threaded Python [experimental]	Skip	80.3 MB

Space Required: 179.1 MB   Remaining: 34 GB

This optional package installs a free-threaded Python framework, including a separate Python interpeter capable of running without the global interpreter lock. It is installed independently of the normal interpeter and includes its own standard library and site-packages. See the installer ReadMe and the release notice for more information.

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>
  ```

  然後 **`activate`**。
- 執行 free-threaded（自由執行緒）版本的 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 使用命令列安裝

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
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/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

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

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)]
```

備F

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 發行 Python 應用程式

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

- py2app：支援從 Python 專案打包成 macOS `.app`。

- Briefcase：BeeWare 專案的一部分；支援建立 macOS .app 的跨平台打包工具，亦🄵管理簽署和驗證 (notarization) 的工具。
- PyInstaller：一個跨平臺打包工具，可以將單一檔案或資料夾打包成可分發的檔案。

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 其他資源

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.

CHAPTER 6

在 Android 上使用 Python

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 `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 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 將 Python 加入 Android 應用程式

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，由 BeeWare 專案提供
- Buildozer，由 Kivy 專案提供
- 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.

- 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`（外部函式庫，例如 OpenSSL）

  - In your assets:
    - `python*.*`（Python 標准函式庫）
    - `python*.*/site-packages`（你自己的 Python 程式碼）
- 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.

CHAPTER 7

在 iOS 上使用 Python

作者
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 `sys.path`. To ensure compliance with App Store policies, an iOS project must post-process any Python packages, converting `.so` 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:
   - 建置選項
     - 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 "$PROJECT_DIR/Python.xcframework/ios-arm64/lib/" "$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 path.
    FULL_MODULE_NAME=$(echo $PYTHON_EXT | cut -d "." -f 1 | tr "/" ".");
    # A bundle identifier; not actually used, but required by Xcode framework packaging
    FRAMEWORK_BUNDLE_ID=$(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")
```

(繼續下一頁)

(繼續上一頁)

```
echo "Install Python $PYTHON_VER standard library extension modules..."
find "$CODESIGNING_FOLDER_PATH/python/lib/$PYTHON_VER/lib-dynload" -name "*.so" |␣
↪while read FULL_EXT; do
   install_dylib python/lib/$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 (`PyPreConfig.utf8_mode`) is *enabled*;
    - Buffered stdio (`PyConfig.buffered_stdio`) is *disabled*;
    - Writing bytecode (`PyConfig.write_bytecode`) is *disabled*;
    - Signal handlers (`PyConfig.install_signal_handlers`) are *enabled*;
    - System logging (`PyConfig.use_system_logger`) is *enabled* (optional, but strongly recommended);
    - `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.

7.2.3 Testing a Python package

The CPython source tree contains a testbed project that is used to run the CPython test suite on the iOS simulator. This testbed can also be used as a testbed project for running your Python library's test suite on iOS.

After building or obtaining an iOS XCFramework (See iOS/README.rst for details), create a clone of the Python iOS testbed project by running:

```
$ python iOS/testbed clone --framework <path/to/Python.xcframework> --app <path/to/module1> -
↪-app <path/to/module2> app-testbed
```

You will need to modify the `iOS/testbed` reference to point to that directory in the CPython source tree; any folders specified with the `--app` flag will be copied into the cloned testbed project. The resulting testbed will be created in the `app-testbed` folder. In this example, the `module1` and `module2` would be importable modules at runtime. If your project has additional dependencies, they can be installed into the `app-testbed/iOSTestbed/app_packages` folder (using `pip install --target app-testbed/iOSTestbed/app_packages` or similar).

You can then use the `app-testbed` folder to run the test suite for your app, For example, if `module1.tests` was the entry point to your test suite, you could run:

```
$ python app-testbed run -- module1.tests
```

This is the equivalent of running `python -m module1.tests` on a desktop Python build. Any arguments after the `--` will be passed to the testbed as if they were arguments to `python -m` on a desktop machine.

You can also open the testbed project in Xcode by running:

```
$ open app-testbed/iOSTestbed.xcodeproj
```

This will allow you to use the full Xcode suite of tools for debugging.

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 8

編輯器與 IDE

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

8.1 IDLE --- Python editor and shell

IDLE is Python’ s Integrated Development and Learning Environment and is generally bundled with Python installs. If you are on Linux and do not have IDLE installed see *Installing IDLE on Linux*. For more information see the IDLE docs.

8.2 Other Editors and IDEs

Python’s community wiki has information submitted by the community on Editors and IDEs. Please go to Python Editors and Integrated Development Environments for a comprehensive list.

APPENDIX A

術語表

>>>
: *互動式* shell 的預設 Python 提示字元。常見於能在直譯器中以互動方式被執行的程式碼範例。

...
: 可以表示：

- 在一個被縮排的程式碼區塊、在一對匹配的左右定界符（delimiter，例如括號、方括號、花括號或三引號）□部，或是在指定一個裝飾器 (decorator) 之後，要輸入程式碼時，*互動式* shell 顯示的預設 Python 提示字元。
- □建常數 `Ellipsis`。

abstract base class（抽象基底類□）
: 抽象基底類□（又稱□ ABC）提供了一種定義介面的方法，作□*duck-typing*（鴨子型□）的補充。其他類似的技術，像是 `hasattr()`，則顯得笨拙或是帶有細微的錯誤（例如使用魔術方法 (magic method)）。ABC □用□擬的 subclass（子類□），它們□不繼承自另一個 class（類□），但仍可被 `isinstance()` 及 `issubclass()` 辨識；請參□ `abc` 模組的□明文件。Python 有許多□建的 ABC，用於資料結構（在 `collections.abc` 模組）、數字（在 `numbers` 模組）、串流（在 `io` 模組）及 import 尋檢器和載入器（在 `importlib.abc` 模組）。你可以使用 `abc` 模組建立自己的 ABC。

annotation（□釋）
: 一個與變數、class 屬性、函式的參數或回傳值相關聯的標□。照慣例，它被用來作□*type hint*（型□提示）。

在執行環境 (runtime)，區域變數的□釋無法被存取，但全域變數、class 屬性和函式的□解，會分□被儲存在模組、class 和函式的 `__annotations__` 特殊屬性中。

請參□*variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，這些章節皆有此功能的□明。關於□釋的最佳實踐方法也請參□ annotations-howto。

argument（引數）
: 呼叫函式時被傳遞給*function*（或*method*）的值。引數有兩種：

- 關鍵字引數 *(keyword argument)*：在函式呼叫中，以識□字（identifier，例如 `name=`）開頭的引數，或是以 `**` 後面 dictionary（字典）□的值被傳遞的引數。例如，`3` 和 `5` 都是以下 `complex()` 呼叫中的關鍵字引數：

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

- 位置引數 *(positional argument)*：不是關鍵字引數的引數。位置引數可在一個引數列表的起始處出現，和（或）作🄵 `*` 之後的*iterable*（可🄵代物件）中的元素被傳遞。例如，`3` 和 `5` 都是以下呼叫中的位置引數：

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

引數會被指定給函式主體中的附名區域變數。關於支配這個指定過程的規則，請參🄵 calls 章節。在語法上，任何運算式都可以被用來表示一個引數；其評估值會被指定給區域變數。

另請參🄵術語表的*parameter*（參數）條目、常見問題中的引數和參數之間的差🄵，以及 **PEP 362**。

asynchronous context manager（非同步情境管理器）
一個可以控制 `async with` 陳述式中所見環境的物件，而它是透過定義 `__aenter__()` 和 `__aexit__()` method（方法）來控制的。由 **PEP 492** 引入。

asynchronous generator（非同步🄵生器）
一個會回傳*asynchronous generator iterator*（非同步🄵生器🄵代器）的函式。它看起來像一個以 `async def` 定義的協程函式 (coroutine function)，但不同的是它包含了 `yield` 運算式，能生成一系列可用於 `async for` 🄵圈的值。

這個術語通常用來表示一個非同步🄵生器函式，但在某些情境中，也可能是表示非同步🄵生器🄵代器 *(asynchronous generator iterator)*。萬一想表達的意思不🄵清楚，那就使用完整的術語，以避免歧義。

一個非同步🄵生器函式可能包含 `await` 運算式，以及 `async for` 和 `async with` 陳述式。

asynchronous generator iterator（非同步🄵生器🄵代器）
一個由*asynchronous generator*（非同步🄵生器）函式所建立的物件。

這是一個*asynchronous iterator*（非同步🄵代器），當它以 `__anext__()` method 被呼叫時，會回傳一個可等待物件 (awaitable object)，該物件將執行非同步🄵生器的函式主體，直到遇到下一個 `yield` 運算式。

每個 `yield` 會暫停處理程序，🄵記住執行狀態（包括區域變數及擱置中的 try 陳述式）。當非同步🄵生器🄵代器以另一個被 `__anext__()` 回傳的可等待物件有效地回復時，它會從停止的地方繼續執行。請參🄵 **PEP 492** 和 **PEP 525**。

asynchronous iterable（非同步可🄵代物件）
一個物件，它可以在 `async for` 陳述式中被使用。必須從它的 `__aiter__()` method 回傳一個*asynchronous iterator*（非同步🄵代器）。由 **PEP 492** 引入。

asynchronous iterator（非同步🄵代器）
一個實作 `__aiter__()` 和 `__anext__()` method 的物件。`__anext__()` 必須回傳一個*awaitable*（可等待物件）。`async for` 會解析非同步🄵代器的 `__anext__()` method 所回傳的可等待物件，直到它引發 `StopAsyncIteration` 例外。由 **PEP 492** 引入。

attribute（屬性）
一個與某物件相關聯的值，該值大多能透過使用點分隔運算式 (dotted expression) 的名稱被參照。例如，如果物件 *o* 有一個屬性 *a*，則該屬性能以 *o.a* 被參照。

如果一個物件允許，給予該物件一個名稱不是由 identifiers 所定義之識🄵符 (identifier) 的屬性是有可能的，例如使用 `setattr()`。像這樣的屬性將無法使用點分隔運算式來存取，而是需要使用 `getattr()` 來取得它。

awaitable（可等待物件）
一個可以在 `await` 運算式中被使用的物件。它可以是一個*coroutine*（協程），或是一個有 `__await__()` method 的物件。另請參🄵 **PEP 492**。

BDFL
Benevolent Dictator For Life（終身仁慈獨裁者），又名 Guido van Rossum，Python 的創造者。

binary file（二進位檔案）
一個能🄵讀取和寫入*bytes-like objects*（類位元組串物件）的*file object*（檔案物件）。二進位檔案的例子有：以二進位模式（`'rb'`、`'wb'` 或 `'rb+'`）開🄵的檔案、`sys.stdin.buffer`、`sys.stdout.buffer`，以及 `io.BytesIO` 和 `gzip.GzipFile` 實例。

另請參🄵*text file*（文字檔案），它是一個能🄵讀取和寫入 `str` 物件的檔案物件。

borrowed reference（借用參照）
在 Python 的 C API 中，借用參照是一個對物件的參照，其中使用該物件的程式碼🄵不擁有這個參照。如果該物件被銷🄵，它會成🄵一個迷途指標 (dangling pointer)。例如，一次垃圾回收 (garbage collection) 可以移除對物件的最後一個*strong reference*（🄵參照），而將該物件銷🄵。

對*borrowed reference* 呼叫 `Py_INCREF()` 以將它原地 (in-place) 轉🄵🄵*strong reference* 是被建議的做法，除非該物件不能在最後一次使用借用參照之前被銷🄵。`Py_NewRef()` 函式可用於建立一個新的*strong reference*。

bytes-like object（類位元組串物件）
一個支援 bufferobjects 且能🄵匯出 C-*contiguous* 緩衝區的物件。這包括所有的 `bytes`、`bytearray` 和 `array.array` 物件，以及許多常見的 `memoryview` 物件。類位元組串物件可用於處理二進位資料的各種運算；這些運算包括壓縮、儲存至二進位檔案和透過 socket（插座）發送。

有些運算需要二進位資料是可變的。🄵明文件通常會將這些物件稱🄵「可讀寫的類位元組串物件」。可變緩衝區的物件包括 `bytearray`，以及 `bytearray` 的 `memoryview`。其他的運算需要讓二進位資料被儲存在不可變物件（「唯讀的類位元組串物件」）中；這些物件包括 `bytes`，以及 `bytes` 物件的 `memoryview`。

bytecode（位元組碼）
Python 的原始碼會被編譯成位元組碼，它是 Python 程式在 CPython 直譯器中的🄵部表示法。該位元組碼也會被暫存在 `.pyc` 檔案中，以便第二次執行同一個檔案時能🄵更快速（可以不用從原始碼重新編譯🄵位元組碼）。這種「中間語言 (intermediate language)」據🄵是運行在一個*virtual machine*（🄵擬機器）上，該🄵擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python 🄵擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在 dis 模組的🄵明文件中找到。

callable（可呼叫物件）
一個 callable 是可以被呼叫的物件，呼叫時可能以下列形式帶有一組引數（請見*argument*）：

```
callable(argument1, argument2, argumentN)
```

一個*function* 與其延伸的*method* 都是 callable。一個有實作 `__call__()` 方法的 class 之實例也是個 callable。

callback（回呼）
作🄵引數被傳遞的一個副程式 (subroutine) 函式，會在未來的某個時間點被執行。

class（類🄵）
一個用於建立使用者定義物件的模板。Class 的定義通常會包含 method 的定義，這些 method 可以在 class 的實例上進行操作。

class variable（類🄵變數）
一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

closure variable（閉包變數）
從外部作用域中定義且從巢狀作用域參照的自由變數，不是於 runtime 從全域或🄵建命名空間解析。可以使用 `nonlocal` 關鍵字明確定義以允許寫入存取，或者如果僅需讀取變數則隱式定義即可。

例如在下面程式碼中的 `inner` 函式中，`x` 和 `print` 都是自由變數，但只有 `x` 是閉包變數：

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

由於 `codeobject.co_freevars` 屬性（🄵管名稱如此，但它僅包含閉包變數的名稱，而不是列出所有參照的自由變數），當預期含義是特指閉包變數時，有時候甚至也會使用更通用的自由變數一詞。

complex number（🄵數）
一個我們熟悉的實數系統的擴充，在此所有數字都會被表示🄵一個實部和一個🄵部之和。🄵數就是🄵數單位（-1 的平方根）的實數倍，此單位通常在數學中被寫🄵 i，在工程學中被寫🄵 j。Python 🄵建了對🄵數的支援，它是用後者的記法來表示🄵數；🄵部會帶著一個後綴的 j 被編寫，例如 3+1j。若要將 math 模組🄵的工具等效地用於🄵數，請使用 cmath 模組。🄵數的使用是一個相當進階的數學功能。如果你🄵有察覺到對它們的需求，那🄵幾乎能確定你可以安全地忽略它們。

context（情境）
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*.
- 一個 `contextvars.Context` 物件。另請參🄵*current context*。

context management protocol（情境管理協定）
由 `with` 陳述式所呼叫的 `__enter__()` 和 `__exit__()` 方法。另請參🄵 **PEP 343**。

context manager（情境管理器）
An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

context variable（情境變數）
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（連續的）
如果一個緩衝區是 *C-contiguous* 或是 *Fortran contiguous*，則它會確切地被視🄵是連續的。零維 (zero-dimensional) 的緩衝區都是 C 及 Fortran contiguous。在一維 (one-dimensional) 陣列中，各項目必須在記憶體中彼此相鄰地排列，而其索引順序是從零開始遞增。在多維的 (multidimensional) C-contiguous 陣列中，按記憶體位址的順序訪問各個項目時，最後一個索引的變化最快。然而，在 Fortran contiguous 陣列中，第一個索引的變化最快。

coroutine（協程）
協程是副程式 (subroutine) 的一種更🄵廣義的形式。副程式是在某個時間點被進入🄵在另一個時間點被退出。協程可以在許多不同的時間點被進入、退出和回復。它們能🄵以 `async def` 陳述式被實作。另請參🄵 **PEP 492**。

coroutine function（協程函式）
一個回傳*coroutine*（協程）物件的函式。一個協程函式能以 `async def` 陳述式被定義，🄵可能會包含 `await`、`async for` 和 `async with` 關鍵字。這些關鍵字由 **PEP 492** 引入。

CPython
Python 程式語言的標准實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 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（裝飾器）
一個函式，它會回傳另一個函式，通常它會使用 `@wrapper` 語法，被應用🄵一種函式的變🄵 (function transformation)。裝飾器的常見範例是 `classmethod()` 和 `staticmethod()`。

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

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

(繼續下一頁)

(繼續上一頁)

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

Class 也存在相同的概念，但在那裡比較不常用。關於裝飾器的更多內容，請參閱函式定義和 class 定義的說明文件。

descriptor（描述器）
任何定義了 `__get__()`、`__set__()` 或 `__delete__()` method 的物件。當一個 class 屬性是一個描述器時，它的特殊連結行為會在屬性查找時被觸發。通常，使用 *a.b* 來取得、設定或刪除某個屬性時，會在 *a* 的 class 字典中查找名稱為 *b* 的物件，但如果 *b* 是一個描述器，則相對應的描述器 method 會被呼叫。對描述器的理解是深入理解 Python 的關鍵，因為它們是許多功能的基礎，這些功能包括函式、method、屬性 (property)、class method、靜態 method，以及對 super class（父類別）的參照。

關於描述器 method 的更多資訊，請參閱 descriptors 或描述器使用指南。

dictionary（字典）
一個關聯陣列 (associative array)，其中任意的鍵會被對映到值。鍵可以是任何帶有 `__hash__()` 和 `__eq__()` method 的物件。在 Perl 中被稱為雜湊 (hash)。

dictionary comprehension（字典綜合運算）
一種緊密的方法，用來處理一個可迭代物件中的全部或部分元素，並將處理結果以一個字典回傳。`results = {n: n ** 2 for n in range(10)}` 會產生一個字典，它包含了鍵 `n` 對映到值 `n ** 2`。請參閱 comprehensions。

dictionary view（字典檢視）
從 `dict.keys()`、`dict.values()` 及 `dict.items()` 回傳的物件被稱為字典檢視。它們提供了字典中項目的動態檢視，這表示當字典有變動時，該檢視會反映這些變動。若要強制將字典檢視轉為完整的 list（串列），須使用 `list(dictview)`。請參閱 dict-views。

docstring（說明字串）
一個在 class、函式或模組中，作為第一個運算式出現的字串文本。雖然它在套件執行時會被忽略，但它會被編譯器辨識，並被放入所屬 class、函式或模組的 `__doc__` 屬性中。由於說明字串可以透過內省 (introspection) 來瀏覽，因此它是物件的說明文件存放的標準位置。

duck-typing（鴨子型別）
一種程式設計風格，它不是藉由檢查一個物件的型別來確定它是否具有正確的介面；取而代之的是，method 或屬性會單純地被呼叫或使用。（「如果它看起來像一隻鴨子而且叫起來像一隻鴨子，那麼它一定是一隻鴨子。」）因為強調介面而非特定型別，精心設計的程式碼能讓多形替代 (polymorphic substitution) 來增進它的靈活性。鴨子型別要避免使用 `type()` 或 `isinstance()` 進行測試。（但是請注意，鴨子型別可以用抽象基底類別 *(abstract base class)* 來補充。）然而，它通常會採用 `hasattr()` 測試，或是*EAFP* 程式設計風格。

EAFP
Easier to ask for forgiveness than permission.（請求寬恕比請求許可更容易。）這種常見的 Python 編碼風格會先假設有效的鍵或屬性的存在，並在該假設被推翻時再捕獲例外。這種乾淨且快速的風格，其特色是存在許多的 `try` 和 `except` 陳述式。該技術與許多其他語言（例如 C）常見的*LBYL* 風格形成了對比。

expression（運算式）
一段可以被評估並求值的語法。換句話說，一個運算式就是文字、名稱、屬性存取、運算子或函式呼叫等運算式元件的累積，而這些元件都能回傳一個值。與許多其他語言不同的是，並非所有的 Python 語言構造都是運算式。另外有一些*statement*（陳述式）不能被用作運算式，例如 `while`。賦值 (assignment) 也是陳述式，而不是運算式。

extension module（擴充模組）
一個以 C 或 C++ 編寫的模組，它使用 Python 的 C API 來與核心及使用者程式碼進行互動。

f-string（f 字串）
以 `'f'` 或 `'F'` 為前綴的字串文本通常被稱為「f 字串」，它是格式化的字串文本的縮寫。另請參閱 **PEP 498**。

file object（檔案物件）

一個讓使用者透過檔案導向 (file-oriented) API（如 `read()` 或 `write()` 等 method）來操作底層資源的物件。根據檔案物件被建立的方式，它能協調對真實磁碟檔案或是其他類型的儲存器或通訊裝置（例如標准輸入 / 輸出、記憶體緩衝區、socket（插座）、管 (pipe) 等）的存取。檔案物件也被稱類檔案物件 *(file-like object)* 或串流 *(stream)*。

實際上，有三種檔案物件：原始的二進位檔案、緩衝的二進位檔案和文字檔案。它們的介面在 `io` 模組中被定義。建立檔案物件的標准方法是使用 `open()` 函式。

file-like object（類檔案物件）

file object（檔案物件）的同義字。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）

Python 所使用的一種編碼和錯誤處理函式，用來解碼來自作業系統的位元組，以及將 Unicode 編碼到作業系統。

檔案系統編碼必須保證能成功解碼所有小於 128 的位元組。如果檔案系統編碼無法提供此保證，則 API 函式會引發 `UnicodeError`。

`sys.getfilesystemencoding()` 和 `sys.getfilesystemencodeerrors()` 函式可用於取得檔案系統編碼和錯誤處理函式。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）會在 Python 動時由 `PyConfig_Read()` 函式來配置：請參 `filesystem_encoding`，以及 `PyConfig` 的成員 `filesystem_errors`。

另請參*locale encoding*（區域編碼）。

finder（尋檢器）

一個物件，它會嘗試正在被 import 的模組尋找*loader*（載入器）。

有兩種類型的尋檢器：元路徑尋檢器 *(meta path finder)* 會使用 `sys.meta_path`，而路徑項目尋檢器 *(path entry finder)* 會使用 `sys.path_hooks`。

請參 finders-and-loaders 和 `importlib` 以了解更多細節。

floor division（向下取整除法）

向下無條件舍去到最接近整數的數學除法。向下取整除法的運算子是 `//`。例如，運算式 `11 // 4` 的計算結果 `2`，與 float（浮點數）真除法所回傳的 `2.75` 不同。請注意，`(-11) // 4` 的結果是 `-3`，因是 `-2.75` 被向下無條件舍去。請參 **PEP 238**。

free threading（自由執行緒）

一種執行緒模型，多個執行緒可以在同一直譯器中同時運行 Python 位元組碼。這與全域直譯器鎖形成對比，後者一次只允許一個執行緒執行 Python 位元組碼。請參 **PEP 703**。

free variable（自由變數）

Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

function（函式）

一連串的陳述式，它能向呼叫者回傳一些值。它也可以被傳遞零個或多個引數，這些引數可被使用於函式本體的執行。另請參*parameter*（參數）、*method*（方法），以及 function 章節。

function annotation（函式釋）

函式參數或回傳值的一個*annotation*（釋）。

函式釋通常被使用於型提示：例如，這個函式預期會得到兩個 `int` 引數，會有一個 `int` 回傳值：

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

函式釋的語法在 function 章節有詳細解釋。

請參🄵*variable annotation* 和 **PEP 484**，皆有此功能的描述。關於🄵釋的最佳實踐方法，另請參🄵 annotations-howto。

__future__
future 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成🄵標准的語法或語義，來編譯當前的模組。而 `__future__` 模組則記🄵了 *feature*（功能）可能的值。透過 import 此模組🄵對其變數求值，你可以看見一個新的功能是何時首次被新增到此語言中，以及它何時將會（或已經）成🄵預設的功能：

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

garbage collection（垃圾回收）
當記憶體不再被使用時，將其釋放的過程。Python 執行垃圾回收，是透過參照計數 (reference counting)，以及一個能🄵檢測和中斷參照循環 (reference cycle) 的循環垃圾回收器 (cyclic garbage collector) 來完成。垃圾回收器可以使用 `gc` 模組對其進行控制。

generator（🄵生器）
一個會回傳*generator iterator*（🄵生器🄵代器）的函式。它看起來像一個正常的函式，但不同的是它包含了 `yield` 運算式，能🄵生一系列的值，這些值可用於 for 🄵圈，或是以 `next()` 函式，每次檢索其中的一個值。

這個術語通常用來表示一個🄵生器函式，但在某些情境中，也可能是表示🄵生器🄵代器。萬一想表達的意思不🄵清楚，那就使用完整的術語，以避免歧義。

generator iterator（🄵生器🄵代器）
一個由*generator*（🄵生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，🄵記住執行狀態（包括區域變數及擱置中的 try 陳述式）。當🄵生器🄵代器回復時，它會從停止的地方繼續執行（與那些每次調用時都要重新開始的函式有所不同）。

generator expression（🄵生器運算式）
一個會回傳🄵代器的運算式。它看起來像一個正常的運算式，後面接著一個 `for` 子句，該子句定義了🄵圈變數、範圍以及一個選擇性的 `if` 子句。該組合運算式會🄵外層函式🄵生多個值：

```
>>> sum(i*i for i in range(10))         # 平方之和 0, 1, 4, ... 81
285
```

generic function（泛型函式）
一個由多個函式組成的函式，該函式會對不同的型🄵實作相同的運算。呼叫期間應該使用哪種實作，是由調度演算法 (dispatch algorithm) 來🄵定。

另請參🄵*single dispatch*（單一調度）術語表條目、`functools.singledispatch()` 裝飾器和 **PEP 443**。

generic type（泛型型🄵）
一個能🄵被參數化 (parameterized) 的*type*（型🄵）；通常是一個 容器型🄵，像是 `list` 和 `dict`。它被用於型🄵提示和🄵釋。

詳情請參🄵泛型🄵名型🄵、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模組。

GIL
請參🄵*global interpreter lock*（全域直譯器鎖）。

global interpreter lock（全域直譯器鎖）
CPython 直譯器所使用的機制，用以確保每次都只有一個執行緒能執行 Python 的*bytecode*（位元組碼）。透過使物件模型（包括關鍵的🄵建型🄵，如 `dict`）自動地避免🄵行存取 (concurrent access) 的危險，此機制可以簡化 CPython 的實作。鎖定整個直譯器，會使直譯器更容易成🄵多執行緒 (multi-threaded)，但代價是會犧牲掉多處理器的機器能🄵提供的一大部分平行性 (parallelism)。

然而，有些擴充模組，無論是標准的或是第三方的，它們被設計成在執行壓縮或雜🄵等計算密集 (computationally intensive) 的任務時，可以解除 GIL。另外，在執行 I/O 時，GIL 總是會被解除。

從 Python 3.13 開始可以使用`--disable-gil` 建置設定來停用 GIL。使用此選項建立 Python 後，必須使用`-X gil=0` 來執行程式碼，或者設定 `PYTHON_GIL=0` 環境變數後再執行程式碼。此功能可以提高多執行緒應用程式的效能，□使多核心 CPU 的高效使用變得更加容易。有關更多詳細資訊，請參□ **PEP 703**。

hash-based pyc（雜□架構的 pyc）
一個位元組碼 (bytecode) 暫存檔，它使用雜□值而不是對應原始檔案的最後修改時間，來確定其有效性。請參□ pyc-invalidation。

hashable（可雜□的）
如果一個物件有一個雜□值，該值在其生命□期中永不改變（它需要一個 `__hash__()` method），且可與其他物件互相比較（它需要一個 `__eq__()` method），那□它就是一個可雜□物件。比較結果□相等的多個可雜□物件，它們必須擁有相同的雜□值。

可雜□性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因□這些資料結構都在其□部使用了雜□值。

大多數的 Python 不可變□建物件都是可雜□的；可變的容器（例如 list 或 dictionary）□不是；而不可變的容器（例如 tuple（元組）和 frozenset），只有當它們的元素是可雜□的，它們本身才是可雜□的。若物件是使用者自定 class 的實例，則這些物件會被預設□可雜□的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜□值則是衍生自它們的 `id()`。

IDLE
Python 的 Integrated Development and Learning Environment（整合開發與學習環境）。idle 是一個基本的編輯器和直譯器環境，它和 Python 的標准發行版本一起被提供。

immortal（不滅）
不滅物件 *(Immortal objects)* 是 **PEP 683** 引入的 CPython 實作細節。

如果一個物件是不滅的，它的參照計數永遠不會被修改，因此在直譯器運行時它永遠不會被釋放。例如，`True` 和 `None` 在 CPython 中是不滅的。

immutable（不可變物件）
一個具有固定值的物件。不可變物件包括數字、字串和 tuple（元組）。這類物件是不能被改變的。如果一個不同的值必須被儲存，則必須建立一個新的物件。它們在需要□定雜□值的地方，扮演重要的角色，例如 dictionary（字典）中的一個鍵。

import path（引入路徑）
一個位置（或路徑項目）的列表，而那些位置就是在 import 模組時，會被*path based finder*（基於路徑的尋檢器）搜尋模組的位置。在 import 期間，此位置列表通常是來自 `sys.path`，但對於子套件 (subpackage) 而言，它也可能是來自父套件的 `__path__` 屬性。

importing（引入）
一個過程。一個模組中的 Python 程式碼可以透過此過程，被另一個模組中的 Python 程式碼使用。

importer（引入器）
一個能□尋找及載入模組的物件；它既是*finder*（尋檢器）也是*loader*（載入器）物件。

interactive（互動的）
Python 有一個互動式直譯器，這表示你可以在直譯器的提示字元輸入陳述式和運算式，立即執行它們□且看到它們的結果。只要□動 `python`，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模塊和包的非常□大的方法（請記住 help(x)）。更多互動式模式相關資訊請見 tut-interac。

interpreted（直譯的）
Python 是一種直譯語言，而不是編譯語言，不過這個區分可能有些模糊，因□有位元組碼 (bytecode) 編譯器的存在。這表示原始檔案可以直接被運行，而不需明確地建立另一個執行檔，然後再執行它。直譯語言通常比編譯語言有更短的開發 / 除錯□期，不過它們的程式通常也運行得較慢。另請參□*interactive*（互動的）。

interpreter shutdown（直譯器關閉）
當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵□部結構。它也會多次呼叫垃圾回收器 *(garbage collector)*。這能□觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，□執行其中的程式碼。在關閉階段被

執行的程式碼會遇到各種例外，因□它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

直譯器關閉的主要原因，是 `__main__` 模組或正被運行的□本已經執行完成。

iterable（可□代物件）

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.

可□代物件可用於 `for` □圈和許多其他需要一個序列的地方 (`zip()`、`map()`...)。當一個可□代物件作□引數被傳遞給□建函式 `iter()` 時，它會□該物件回傳一個□代器。此□代器適用於針對一組值進行一遍 (one pass) 運算。使用□代器時，通常不一定要呼叫 `iter()` 或自行處理□代器物件。`for` 陳述式會自動地□你處理這些事，它會建立一個暫時性的未命名變數，用於在□圈期間保有該□代器。另請參□*iterator*（□代器）、*sequence*（序列）和*generator*（□生器）。

iterator（□代器）

一個表示資料流的物件。重□地呼叫□代器的 `__next__()` method（或是將它傳遞給□建函式 `next()`）會依序回傳資料流中的各項目。當不再有資料時，則會引發 `StopIteration` 例外。此時，該□代器物件已被用盡，而任何對其 `__next__()` method 的進一步呼叫，都只會再次引發 `StopIteration`。□代器必須有一個 `__iter__()` method，它會回傳□代器物件本身，所以每個□代器也都是可□代物件，且可以用於大多數適用其他可□代物件的場合。一個明顯的例外，是嘗試多遍□代 (multiple iteration passes) 的程式碼。一個容器物件（像是 `list`）在每次你將它傳遞給 `iter()` 函式或在 `for` □圈中使用它時，都會□生一個全新的□代器。使用□代器嘗試此事（多遍□代）時，只會回傳在前一遍□代中被用過的、同一個已被用盡的□代器物件，使其看起來就像一個空的容器。

在 typeiter 文中可以找到更多資訊。

CPython □不是始終如一地都會檢查「□代器有定義 `__iter__()`」這個規定。另請注意，free-threading（自由執行緒）CPython 不保證□代器操作的執行緒安全。

key function（鍵函式）

鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，`locale.strxfrm()` 被用來□生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括 `min()`、`max()`、`sorted()`、`list.sort()`、`heapq.merge()`、`heapq.nsmallest()`、`heapq.nlargest()` 和 `itertools.groupby()`。

有幾種方法可以建立一個鍵函式。例如，`str.lower()` method 可以作□不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 `lambda` 運算式被建造，例如 `lambda r: (r[0], r[2])`。另外，`operator.attrgetter()`、`operator.itemgetter()` 和 `operator.methodcaller()` □三個鍵函式的建構函式 (constructor)。關於如何建立和使用鍵函式的範例，請參□如何排序。

keyword argument（關鍵字引數）

請參□*argument*（引數）。

lambda

由單一*expression*（運算式）所組成的一個匿名行□函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 `lambda [parameters]: expression`

LBYL

Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先□條件。這種風格與*EAFP* 方式形成對比，且它的特色是會有許多 `if` 陳述式的存在。

在一個多執行緒環境中，LBYL 方式有在「三思」和「後行」之間引入了競□條件 (race condition) 的風險。例如以下程式碼 `if key in mapping: return mapping[key]`，如果另一個執行緒在測試之後但在查找之前，從 *mapping* 中移除了 *key*，則該程式碼就會失效。這個問題可以用鎖 (lock) 或使用 EAFP 編碼方式來解□。

list（串列）

一個 Python □建的*sequence*（序列）。□管它的名字是 list，它其實更類似其他語言中的一個陣列

(array) 而較不像一個鏈結串列 (linked list)，因🄵存取元素的時間🄵雜度是 $O(1)$。

list comprehension（串列綜合運算）
一種用來處理一個序列中的全部或部分元素，🄵將處理結果以一個 list 回傳的簡要方法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 會🄵生一個字串 list，其中包含 0 到 255 範圍🄵，所有偶數的十六進位數 (0x..)。`if` 子句是選擇性的。如果省略它，則 `range(256)` 中的所有元素都會被處理。

loader（載入器）
一個能🄵載入模組的物件。它必須定義 `exec_module()` 和 `create_module()` 方法以實作 `Loader` 介面。載入器通常是被*finder*（尋檢器）回傳。更多細節請參🄵：

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

locale encoding（區域編碼）
在 Unix 上，它是 LC_CTYPE 區域設定的編碼。它可以用 `locale.setlocale(locale.LC_CTYPE, new_locale)` 來設定。

在 Windows 上，它是 ANSI 代碼頁（code page，例如 `"cp1252"`）。

在 Android 和 VxWorks 上，Python 使用 `"utf-8"` 作🄵區域編碼。

`locale.getencoding()` 可以用來取得區域編碼。

也請參考*filesystem encoding and error handler*。

magic method（魔術方法）
special method（特殊方法）的一個非正式同義詞。

mapping（對映）
一個容器物件，它支援任意鍵的查找，且能實作 abstract base classes（抽象基底類🄵）中，`collections.abc.Mapping` 或 `collections.abc.MutableMapping` 所指定的 method。範例包括 `dict`、`collections.defaultdict`、`collections.OrderedDict` 和 `collections.Counter`。

meta path finder（元路徑尋檢器）
一種經由搜尋 `sys.meta_path` 而回傳的*finder*（尋檢器）。元路徑尋檢器與*路徑項目尋檢器 (path entry finder)* 相關但是不同。

關於元路徑尋檢器實作的 method，請參🄵 `importlib.abc.MetaPathFinder`。

metaclass（元類🄵）
一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個 base class（基底類🄵）的列表。Metaclass 負責接受這三個引數，🄵建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特🄵之處在於它能🄵建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供🄵大且優雅的解🄵方案。它們已被用於記🄵屬性存取、增加執行緒安全性、追🄵物件建立、實作單例模式 (singleton)，以及許多其他的任務。

更多資訊可以在 metaclasses 章節中找到。

method（方法）
一個在 class 本體🄵被定義的函式。如果 method 作🄵其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成🄵它的第一個*argument*（引數）（此引數通常被稱🄵 `self`）。請參🄵*function*（函式）和*nested scope*（巢狀作用域）。

method resolution order（方法解析順序）
方法解析順序是在查找某個成員的過程中，base class（基底類🄵）被搜尋的順序。關於 Python 自 2.3 版直譯器所使用的演算法細節，請參🄵 python_2.3_mro。

module（模組）
一個擔任 Python 程式碼的組織單位 (organizational unit) 的物件。模組有一個命名空間，它包含任意的 Python 物件。模組是藉由*importing* 的過程，被載入至 Python。

另請參🄵*package*（套件）。

module spec（模組規格）

一個命名空間，它包含用於載入模組的 import 相關資訊。它是 `importlib.machinery.ModuleSpec` 的一個實例。

另請參🄵 module-specs。

MRO

請參🄵*method resolution order*（方法解析順序）。

mutable（可變物件）

可變物件可以改變它們的值，但維持它們的 `id()`。另請參🄵*immutable*（不可變物件）。

named tuple（附名元組）

術語「named tuple（附名元組）」是指從 tuple 繼承的任何型🄵或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型🄵或 class 也可以具有其他的特性。

有些🄵建型🄵是 named tuple，包括由 `time.localtime()` 和 `os.stat()` 回傳的值。另一個例子是 `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
```

有些 named tuple 是🄵建型🄵（如上例）。或者，一個 named tuple 也可以從一個正規的 class 定義來建立，只要該 class 是繼承自 `tuple`，且定義了附名欄位 (named field) 即可。這類的 class 可以手工編寫、可以繼承自 `typing.NamedTuple` 來建立，也可以使用工廠函式 (factory function) `collections.namedtuple()` 來建立。後者技術也增加了一些額外的 method，這些 method 可能是在手寫或🄵建的 named tuple 中，無法找到的。

namespace（命名空間）

變數被儲存的地方。命名空間是以 dictionary（字典）被實作。有區域的、全域的及🄵建的命名空間，而在物件中（在 method 中）也有巢狀的命名空間。命名空間藉由防止命名衝突，來支援模組化。例如，函式 `builtins.open` 和 `os.open()` 是透過它們的命名空間來區分彼此。命名空間也藉由明確地區分是哪個模組在實作一個函式，來增進可讀性及可維護性。例如，寫出 `random.seed()` 或 `itertools.islice()` 明確地表示，這些函式分🄵是由 `random` 和 `itertools` 模組在實作。

namespace package（命名空間套件）

A *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.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and reference-namespace-package.

另請參🄵*module*（模組）。

nested scope（巢狀作用域）

能🄵參照外層定義 (enclosing definition) 中的變數的能力。舉例來🄵，一個函式如果是在另一個函式中被定義，則它便能🄵參照外層函式中的變數。請注意，在預設情🄵下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最🄵層作用域中讀取及寫入。同樣地，全域變數是在全域命名空間中讀取及寫入。`nonlocal` 容許對外層作用域進行寫入。

new-style class（新式類🄵）

一個舊名，它是指現在所有的 class 物件所使用的 class 風格。在早期的 Python 版本中，只有新式 class 才能使用 Python 較新的、多樣的功能，像是 `__slots__`、描述器 (descriptor)、屬性 (property)、`__getattribute__()`、class method（類🄵方法）和 static method（🄵態方法）。

object（物件）

具有狀態（屬性或值）及被定義的行🄵（method）的任何資料。它也是任何*new-style class*（新式類🄵）的最終 base class（基底類🄵）。

optimized scope（最佳化作用域）
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（套件）
一個 Python 的*module*（模組），它可以包含子模組 (submodule) 或是遞F的子套件 (subpackage)。技術上而言，套件就是具有 `__path__` 屬性的一個 Python 模組。

另請參F*regular package*（正規套件）和*namespace package*（命名空間套件）。

parameter（參數）
在*function*（函式）或 method 定義中的一個命名實體 (named entity)，它指明該函式能F接受的一個*argument*（引數），或在某些情F下指示多個引數。共有有五種不同的參數類型：

- *positional-or-keyword*（位置或關鍵字）：指明一個可以按照位置或是作F關鍵字引數被傳遞的引數。這是參數的預設類型，例如以下的 *foo* 和 *bar*：

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

- *positional-only*（僅限位置）：指明一個只能按照位置被提供的引數。在函式定義的參數列表中包含一個 / 字元，就可以在該字元前面定義僅限位置參數，例如以下的 *posonly1* 和 *posonly2*：

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

- *keyword-only*（僅限關鍵字）：指明一個只能以關鍵字被提供的引數。在函式定義的參數列表中，包含一個任意數量位置參數 (var-positional parameter) 或是單純的 * 字元，就可以在其後方定義僅限關鍵字參數，例如以下的 *kw_only1* 和 *kw_only2*：

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

- *var-positional*（任意數量位置）：指明一串能以任意序列被提供的位置引數（在已被其他參數接受的任何位置引數之外）。這類參數是透過在其參數名稱字首加上 * 來定義的，例如以下的 *args*：

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

- *var-keyword*（任意數量關鍵字）：指明可被提供的任意數量關鍵字引數（在已被其他參數接受的任何關鍵字引數之外）。這類參數是透過在其參數名稱字首加上 ** 來定義的，例如上面範例中的 *kwargs*。

參數可以指明引數是選擇性的或必需的，也可以F一些選擇性的引數指定預設值。

另請參F術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差F、`inspect.Parameter` class、function 章節，以及 **PEP 362**。

path entry（路徑項目）
在*import path*（引入路徑）中的一個位置，而*path based finder*（基於路徑的尋檢器）會參考該位置來尋找要 import 的模組。

path entry finder（路徑項目尋檢器）
被 `sys.path_hooks` 中的一個可呼叫物件 (callable)（意即一個*path entry hook*）所回傳的一種*finder*，它知道如何以一個*path entry*定位模組。

關於路徑項目尋檢器實作的 method，請參F `importlib.abc.PathEntryFinder`。

path entry hook（路徑項目F）
在 `sys.path_hooks` 列表中的一個可呼叫物件 (callable)，若它知道如何在一個特定的*path entry* 中尋找模組，則會回傳一個*path entry finder*（路徑項目尋檢器）。

path based finder（基於路徑的尋檢器）
預設的元路徑尋檢器 *(meta path finder)* 之一，它會在一個*import path* 中搜尋模組。

path-like object（類路徑物件）

一個表示檔案系統路徑的物件。類路徑物件可以是一個表示路徑的 `str` 或 `bytes` 物件，或是一個實作 `os.PathLike` 協定的物件。透過呼叫 `os.fspath()` 函式，一個支援 `os.PathLike` 協定的物件可以被轉□□ `str` 或 `bytes` 檔案系統路徑；而 `os.fsdecode()` 及 `os.fsencode()` 則分□可用於確保 `str` 及 `bytes` 的結果。由 **PEP 519** 引入。

PEP

Python Enhancement Proposal（Python 增□提案）。PEP 是一份設計□明文件，它能□ Python 社群提供資訊，或是描述 Python 的一個新功能或該功能的程序和環境。PEP 應該要提供簡潔的技術規範以及被提案功能的運作原理。

PEP 的存在目的，是要成□重大新功能的提案、社群中關於某個問題的意見收集，以及已納入 Python 的設計□策的記□，這些過程的主要機制。PEP 的作者要負責在社群□建立共識□記□反對意見。

請參□ **PEP 1**。

portion（部分）

在單一目□中的一組檔案（也可能儲存在一個 zip 檔中），這些檔案能對一個命名空間套件 (namespace package) 有所貢獻，如同 **PEP 420** 中的定義。

positional argument（位置引數）

請參□*argument*（引數）。

provisional API（暫行 API）

暫行 API 是指，從標准函式庫的向後相容性 (backwards compatibility) 保證中，故意被排除的 API。雖然此類介面，只要它們被標示□暫行的，理論上□不會有重大的變更，但如果核心開發人員認□有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更□不會無端地□生——只有 API 被納入之前未察覺的嚴重基本缺陷被揭露時，它們才會發生。

即使對於暫行 API，向後不相容的變更也會被視□「最後的解□方案」——對於任何被發現的問題，仍然會盡可能找出一個向後相容的解□方案。

這個過程使得標准函式庫能隨著時間不斷進化，而避免耗費過長的時間去鎖定有問題的設計錯誤。請參□ **PEP 411** 了解更多細節。

provisional package（暫行套件）

請參□*provisional API*（暫行 API）。

Python 3000

Python 3.x 系列版本的□稱（很久以前創造的，當時第 3 版的發布是在□遠的未來。）也可以縮寫□「Py3k」。

Pythonic（Python 風格的）

一個想法或一段程式碼，它應用了 Python 語言最常見的慣用語，而不是使用其他語言常見的概念來實作程式碼。例如，Python 中常見的一種習慣用法，是使用一個 `for` 陳述式，對一個可□代物件的所有元素進行□圈。許多其他語言□□有這種類型的架構，所以不熟悉 Python 的人有時會使用一個數值計數器來代替：

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

相較之下，以下方法更簡潔、更具有 Python 風格：

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

qualified name（限定名稱）

一個「點分隔名稱」，它顯示從一個模組的全域作用域到該模組中定義的 class、函式或 method 的「路徑」，如 **PEP 3155** 中的定義。對於頂層的函式和 class 而言，限定名稱與其物件名稱相同：

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

(繼續下一頁)

(繼續上一頁)

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

當用於引用模組時，完全限定名稱 *(fully qualified name)* 是表示該模組的完整點分隔路徑，包括任何的父套件，例如 `email.mime.text`：

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

reference count（參照計數）
對於一個物件的參照次數。當一個物件的參照計數下降到零時，它會被解除配置 (deallocated)。有些物件是「*不滅的* (immortal)」F擁有不會被改變的參照計數，也因此永遠不會被解除配置。參照計數通常在 Python 程式碼中看不到，但它F是*CPython* 實作的一個關鍵元素。程式設計師可以呼叫 `getrefcount()` 函式來回傳一個特定物件的參照計數。

regular package（正規套件）
一個傳統的*package*（套件），例如一個包含 `__init__.py` 檔案的目F。

另請參F*namespace package*（命名空間套件）。

REPL
「read-eval-print F圈 (read–eval–print loop)」的縮寫，是*互動式*直譯器 shell 的另一個名稱。

__slots__
在 class F部的一個宣告，它藉由預先宣告實例屬性的空間，以及消除實例 dictionary（字典），來節省記憶體。雖然該技術很普遍，但它有點難以正確地使用，最好保留給那種在一個記憶體關鍵 (memory-critical) 的應用程式中存在大量實例的罕見情F。

sequence（序列）
一個*iterable*（可F代物件），它透過 `__getitem__()` special method（特殊方法），使用整數索引來支援高效率的元素存取，F定義了一個 `__len__()` method 來回傳該序列的長度。一些F建序列型F包括 `list`、`str`、`tuple` 和 `bytes`。請注意，雖然 `dict` 也支援 `__getitem__()` 和 `__len__()`，但它被視F對映 (mapping) 而不是序列，因F其查找方式是使用任意的*hashable* 鍵，而不是整數。

抽象基底類F (abstract base class) `collections.abc.Sequence` 定義了一個更加豐富的介面，F不僅止於 `__getitem__()` 和 `__len__()`，還增加了 `count()`、`index()`、`__contains__()` 和 `__reversed__()`。實作此擴充介面的型F，可以使用 `register()` 被明確地FF。更多關於序列方法的文件，請見常見序列操作。

set comprehension（集合綜合運算）
一種緊密的方法，用來處理一個可F代物件中的全部或部分元素，F將處理結果以一個 set 回傳。`results = {c for c in 'abracadabra' if c not in 'abc'}` 會F生一個字串 set：`{'r', 'd'}`。請參F comprehensions。

single dispatch（單一調度）
generic function（泛型函式）調度的一種形式，在此，實作的選擇是基於單一引數的型F。

slice（切片）
一個物件，它通常包含一段*sequence*（序列）的某一部分。建立一段切片的方法是使用下標符號 (subscript notation) `[]`，若要給出多個數字，則在數字之間使用冒號，例如 `variable_name[1:3:5]`。在括號（下標）符號的F部，會使用 `slice` 物件。

soft deprecated（軟性F用）
被軟性F用的 API 代表不應再用於新程式碼中，但在現有程式碼中繼續使用它仍會是安全的。API 仍會以文件記FF會被測試，但不會被繼續改進。

與正常🄵用不同，軟性🄵用🄵有🄵除 API 的規劃，也不會發出警告。

請參🄵 PEP 387：軟性🄵用。

special method（特殊方法）
一種會被 Python 自動呼叫的 method，用於對某種型🄵執行某種運算，例如加法。這種 method 的名稱會在開頭和結尾有兩個下底🄵。Special method 在 specialnames 中有詳細🄵明。

statement（陳述式）
陳述式是一個套組（suite，一個程式碼「區塊」）中的一部分。陳述式可以是一個*expression*（運算式），或是含有關鍵字（例如 `if`、`while` 或 `for`）的多種結構之一。

static type checker（🄵態型🄵檢查器）
會讀取 Python 程式碼🄵分析的外部工具，能🄵找出錯誤，像是使用了不正確的型🄵。另請參🄵型🄵提示 *(type hints)* 以及 `typing` 模組。

strong reference（🄵參照）
在 Python 的 C API 中，🄵參照是對物件的參照，該物件🄵持有該參照的程式碼所擁有。建立參照時透過呼叫 `Py_INCREF()` 來獲得🄵參照、🄵除參照時透過 `Py_DECREF()` 釋放🄵參照。

`Py_NewRef()` 函式可用於建立一個對物件的🄵參照。通常，在退出🄵參照的作用域之前，必須在該🄵參照上呼叫 `Py_DECREF()` 函式，以避免🄵漏一個參照。

另請參🄵*borrowed reference*（借用參照）。

text encoding（文字編碼）
Python 中的字串是一個 Unicode 碼點 (code point) 的序列（範圍在 `U+0000` -- `U+10FFFF` 之間）。若要儲存或傳送一個字串，它必須被序列化🄵一個位元組序列。

將一個字串序列化🄵位元組序列，稱🄵「編碼」，而從位元組序列重新建立該字串則稱🄵「解碼 (decoding)」。

有多種不同的文字序列化編解碼器 (codecs)，它們被統稱🄵「文字編碼」。

text file（文字檔案）
一個能🄵讀取和寫入 `str` 物件的一個*file object*（檔案物件）。通常，文字檔案實際上是存取位元組導向的資料流 (byte-oriented datastream) 🄵會自動處理*text encoding*（文字編碼）。文字檔案的例子有：以文字模式（`'r'` 或 `'w'`）開🄵的檔案、`sys.stdin`、`sys.stdout` 以及 `io.StringIO` 的實例。

另請參🄵*binary file*（二進位檔案），它是一個能🄵讀取和寫入類位元組串物件 *(bytes-like object)* 的檔案物件。

triple-quoted string（三引號🄵字串）
由三個雙引號 (") 或單引號 (') 的作🄵邊界的一個字串。雖然它們🄵🄵有提供🄵於單引號字串的任何額外功能，但基於許多原因，它們仍是很有用的。它們讓你可以在字串中包含未跳🄵 (unescaped) 的單引號和雙引號，而且它們不需使用連續字元 (continuation character) 就可以跨越多行，這使得它們在編寫🄵明字串時特🄵有用。

type（型🄵）
一個 Python 物件的型🄵🄵定了它是什🄵類型的物件；每個物件都有一個型🄵。一個物件的型🄵可以用它的 `__class__` 屬性來存取，或以 `type(obj)` 來檢索。

type alias（型🄵🄵名）
一個型🄵的同義詞，透過將型🄵指定給一個識🄵符 (identifier) 來建立。

型🄵🄵名對於簡化型🄵提示 *(type hint)* 很有用。例如：

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

可以寫成這樣，更具有可讀性：

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

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

請參□ `typing` 和 **PEP 484**，有此功能的描述。

type hint（型□提示）
一種*annotation*（□釋），它指定一個變數、一個 class 屬性或一個函式的參數或回傳值的預期型□。

型□提示是選擇性的，而不是被 Python □制的，但它們對□態型□檢查器 *(static type checkers)*很有用，□能協助 IDE 完成程式碼的補全 (completion) 和重構 (refactoring)。

全域變數、class 屬性和函式（不含區域變數）的型□提示，都可以使用 `typing.get_type_hints()` 來存取。

請參□ `typing` 和 **PEP 484**，有此功能的描述。

universal newlines（通用□行字元）
一種解譯文字流 (text stream) 的方式，會將以下所有的情□識□□一行的結束：Unix 行尾慣例 `'\n'`、Windows 慣例 `'\r\n'` 和舊的 Macintosh 慣例 `'\r'`。請參□ **PEP 278** 和 **PEP 3116**，以及用於 `bytes.splitlines()` 的附加用途。

variable annotation（變數□釋）
一個變數或 class 屬性的*annotation*（□釋）。

□釋變數或 class 屬性時，賦值是選擇性的：

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

變數□釋通常用於型□提示 *(type hint)*：例如，這個變數預期會取得 `int`（整數）值：

```
count: int = 0
```

變數□釋的語法在 annassign 章節有詳細的解釋。

請參□*function annotation*（函式□釋）、**PEP 484** 和 **PEP 526**，皆有此功能的描述。關於□釋的最佳實踐方法，另請參□ annotations-howto。

virtual environment（□擬環境）
一個協作隔離 (cooperatively isolated) 的執行環境，能讓 Python 的使用者和應用程式得以安裝和升級 Python 發□套件，而不會對同一個系統上運行的其他 Python 應用程式的行□□生干擾。

另請參□ `venv`。

virtual machine（□擬機器）
一部完全由軟體所定義的電腦 (computer)。Python 的□擬機器會執行由*bytecode*（位元組碼）編譯器所發出的位元組碼。

Zen of Python（Python 之□）
Python 設計原則與哲學的列表，其□容有助於理解和使用此語言。此列表可以透過在互動式提示字元後輸入「`import this`」來找到它。

APPENDIX B

關於這份🄵明文件

Python 🄵明文件是透過使用 Sphinx（一個原🄵 Python 而生的文件🄵生器、目前是以獨立專案形式來維護）將使用 reStructuredText 撰寫的原始檔轉🄵而成。

如同 Python 自身，透過自願者的努力下🄵出文件與封裝後自動化執行工具。若想要回報臭蟲，請見 reporting-bugs 頁面，🄵含相關資訊。我們永遠歡迎新的自願者加入！

致謝：

- Fred L. Drake, Jr.，原始 Python 文件工具集的創造者以及一大部份🄵容的作者；
- 創造 reStructuredText 和 Docutils 工具組的 Docutils 專案；
- Fredrik Lundh 先生，Sphinx 從他的 Alternative Python Reference 計劃中獲得許多的好主意。

B.1 Python 文件的貢獻者們

許多人都曾🄵 Python 這門語言、Python 標准函式庫和 Python 🄵明文件貢獻過。Python 所發🄵的原始碼中含有部份貢獻者的清單，請見 Misc/ACKS 。

正因🄵 Python 社群的撰寫與貢獻才有這份這🄵棒的🄵明文件 -- 感謝所有貢獻過的人們！

APPENDIX C

沿革與授權

C.1 軟體沿革

Python 是由荷蘭數學和計算機科學研究學會（CWI，見 https://www.cwi.nl）的 Guido van Rossum 於 1990 年代早期所創造，目的是作F一種稱F ABC 語言的後繼者。F管 Python 包含了許多來自其他人的貢獻，Guido 仍是其主要作者。

1995 年，Guido 在維吉尼亞州雷斯頓的國家創新研究公司（CNRI，見 https://www.cnri.reston.va.us）繼續他在 Python 的工作，F在那F發F了該軟體的多個版本。

2000 年五月，Guido 和 Python 核心開發團隊轉移到 BeOpen.com F成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations，後來成F Zope Corporation。2001 年，Python 軟體基金會（PSF，見 https://www.python.org/psf/）成立，這是一個專F擁有 Python 相關的智慧財F權而創立的非營利組織。Zope Corporation 過去是 PSF 的一個贊助會員。

所有的 Python 版本都是開源的（有關開源的定義，參F https://opensource.org）。歷史上，大多數但非全部的 Python 版本，也是 GPL 相容的；以下表格總結各個版本的差F。

發F版本	源自	年份	擁有者	GPL 相容性？(1)
0.9.0 至 1.2	不適用	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	是 (2)
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 以上	2.1.1	2001 至今	PSF	是

備F

(1) GPL 相容F不表示我們是在 GPL 下發F Python。不像 GPL，所有的 Python 授權都可以讓你發F修改後的版本，但不一定要使你的變更成F開源。GPL 相容的授權使得 Python 可以結合其他

在 GPL 下發🄵的軟體一起使用；但其它的授權則不行。

(2) 根據 Richard Stallman 的🄵法，1.6.1 不是 GPL 相容的，因🄵其授權有一個法律選擇條款。然而根據 CNRI 的🄵法，Stallman 的律師告訴 CNRI 的律師，1.6.1 與 GPL「不相容」。

感謝許多的外部志工，在 Guido 指導下的付出，使得這些版本的發🄵成🄵可能。

C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和🄵明文件的授權是基於 Python 軟體基金會授權第二版 (Python Software Foundation License Version 2)。

從 Python 3.8.6 開始，🄵明文件中的範例、程式庫和其他程式碼，是被雙重授權 (dual licensed) 於 PSF 授權第二版以及*Zero-Clause BSD 授權*。

有些被納入 Python 中的軟體是基於不同的授權。這些授權將會與其授權之程式碼一起被列出。關於這些授權的不完整清單，請參🄵被收🄵軟體的授權與致謝。

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

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

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

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

4. PSF is making Python available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

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

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

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

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

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
```

(繼續下一頁)

(繼續上一頁)

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

(繼續下一頁)

(繼續上一頁)

```
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 DOCUMENTATION

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

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

C.3 被收🄵軟體的授權與致謝

本節是一個不完整但持續增加的授權與致謝清單，對象是在 Python 發🄵版本中所收🄵的第三方軟體。

C.3.1 Mersenne Twister

`random` 模組底下的 `_random` C 擴充程式包含了以 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 的下載🄵容🄵基礎的程式碼。以下是原始程式碼的完整聲明：

```
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
```

(繼續下一頁)

(繼續上一頁)

C.3.2 Sockets

socket 模組使用 getaddrinfo() 和 getnameinfo() 函式，它們在 WIDE 專案（https://www.wide.ad.jp/）中，於不同的原始檔案中被編碼：

C.3.3 非同步 socket 服務

test.support.asynchat 和 test.support.asyncore 模組包含以下聲明：

(繼續下一頁)

(繼續上一頁)

```
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 管理

`http.cookies` 模組包含以下聲明:

```
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 執行追🄵

`trace` 模組包含以下聲明:

```
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,
```

(繼續下一頁)

(繼續上一頁)

```
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 與 UUdecode 函式

uu 編解碼器包含以下聲明：

C.3.7 XML 遠端程序呼叫

xmlrpc.client 模組包含以下聲明：

(繼續下一頁)

(繼續上一頁)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test.test_epoll` 模組包含以下聲明:

C.3.9 Select kqueue

`select` 模組對於 kqueue 介面包含以下聲明:

C.3.10 SipHash24

`Python/pyhash.c` 檔案包含 Marek Majkowski' 基於 Dan Bernstein 的 SipHash24 演算法的實作。它包含以下聲明：

```
<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 與 dtoa

`Python/dtoa.c` 檔案提供了 C 的 dtoa 和 strtod 函式，用於將 C 的雙精度浮點數和字串互相轉F。該檔案是衍生自 David M. Gay 建立的同名檔案，後者現在可以從 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 下載。於 2009 年 3 月 16 日所檢索的原始檔案包含以下版權與授權聲明：

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

C.3.12 OpenSSL

如果 OpenSSL 函式庫可被作業系統使用，則 `hashlib`、`posix`、`ssl` 模組會使用它來提升效能。此外，因F Windows 和 macOS 的 Python 安裝程式可能包含 OpenSSL 函式庫的副本，所以我們也在此收F OpenSSL 授權的副本。對於 OpenSSL 3.0 版本以及由此衍生的更新版本則適用 Apache 許可證 v2：

```
                              Apache License
                        Version 2.0, January 2004
                     https://www.apache.org/licenses/
```

(繼續下一頁)

(繼續上一頁)

```
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.
```

(繼續下一頁)

(繼續上一頁)

```
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
```

(繼續下一頁)

(繼續上一頁)

```
   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

除非在建置 `pyexpat` 擴充時設定了 `--with-system-expat`，否則該擴充會用一個包含 expat 原始碼的副本來建置：

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

(繼續下一頁)

(繼續上一頁)

```
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

除非在建置 `_ctypes` 模組底下 `_ctypes` 擴充程式時設定🄵 `--with-system-libffi`，否則該擴充會用一個🄵含 libffi 原始碼的副本來建置：

```
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

如果在系統上找到的 zlib 版本太舊以致於無法用於建置 `zlib` 擴充，則該擴充會用一個🄵含 zlib 原始碼的副本來建置：

```
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
```

(繼續下一頁)

(繼續上一頁)

```
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

`tracemalloc` 使用的雜🄵表 (hash table) 實作，是以 cfuhash 專案🄵基礎：

```
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

除非在建置 `decimal` 模組底下 `_decimal` C 擴充程式時設定🄵 `--with-system-libmpdec`，否則該模組會用一個🄵含 libmpdec 函式庫的副本來建置：

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

(繼續下一頁)

(繼續上一頁)

C.3.18 W3C C14N 測試套件

`test` 程式包中的 C14N 2.0 測試套件 (`Lib/test/xmltestdata/c14n-20/`) 是從 W3C 網站 https://www.w3.org/TR/xml-c14n2-testcases/ 被檢索，且是基於 3-clause BSD 授權被發F：

C.3.19 mimalloc

MIT 授權：

C.3.20 asyncio

`asyncio` 模組的部分內容是從 uvloop 0.16 中收錄過來，其基於 MIT 授權來發佈：

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:

(繼續下一頁)

(繼續上一頁)

APPENDIX D

版權宣告

Python 和這份說明文件的版權：

完整的授權條款資訊請參見沿革與授權。

索引

非依字母順序