1. Command line and environment

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

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

1.1. Command line

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

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

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

```
python myscript.py
```

1.1.1. Interface options

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

- When called with standard input connected to a tty device, it prompts for commands and executes them until an EOF (an end-of-file character, you can produce that with `Ctrl-D` on UNIX or `Ctrl-Z, Enter` on Windows) is read.
- 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).

-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.

> **Note:** 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`.

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

```
python -mtimeit -s 'setup here' 'benchmarked code here'
python -mtimeit -h # for details
```

> **See also:**
>
> `runpy.run_module()`
> Equivalent functionality directly available to Python code
>
> **PEP 338** – Executing modules as scripts

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

Changed in version 3.4: namespace packages are also supported

`-`

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

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

`<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.

See also:

`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 Readline configuration).

See also: Invoking the Interpreter

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

1.1.2. Generic options

-?
-h
--help

Print a short description of all command line options.

-V
--version

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

```
Python 3.6.0b2+
```

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

```
Python 3.6.0b2+ (3.6:84a3c5003510+, Oct 26 2016, 02:33:55)
[GCC 6.2.0 20161005]
```

New in version 3.6: The `-VV` option.

1.1.3. Miscellaneous options

-b

Issue a warning when comparing `bytes` or `bytearray` with `str` or `bytes` with `int`. Issue an error when the option is given twice (`-bb`).

Changed in version 3.5: Affects 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`.

-d

Turn on parser debugging output (for wizards only, depending on compilation options). See also `PYTHONDEBUG`.

-E

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

-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 and -s. 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.

New in version 3.4.

-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.

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

-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**).

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

-q

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

New in version 3.2.

-R

Kept for compatibility. On Python 3.3 and greater, hash randomization is turned on by default.

On previous versions of Python, this option turns on hash randomization, so that the __hash__() values of str, bytes and datetime 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://www.ocert.org/advisories/ocert-2011-003.html for details.

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

New in version 3.2.3.

-s

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

> **See also:** **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 binary layer of the stdout and stderr streams (which is available as their `buffer` attribute) to be unbuffered. The text I/O layer will still be line-buffered if writing to the console, or block-buffered if redirected to a non-interactive file.

See also `PYTHONUNBUFFERED`.

-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. See also `PYTHONVERBOSE`.

-W `arg`

Warning control. Python's warning machinery by default prints warning messages to `sys.stderr`. A typical warning message has the following form:

```
file:line: category: message
```

By default, each warning is printed once for each source line where it occurs. This option controls how often warnings are printed.

Multiple `-W` options may 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 from within a Python program using the `warnings` module.

The simplest form of argument is one of the following action strings (or a unique abbreviation):

`ignore`
: Ignore all warnings.

`default`
: Explicitly request the default behavior (printing each warning once per source line).

`all`
: Print a warning each time it occurs (this may generate many messages if a warning is triggered repeatedly for the same source line, such as inside a loop).

`module`
: Print each warning only the first time it occurs in each module.

`once`
: Print each warning only the first time it occurs in the program.

`error`
: Raise an exception instead of printing a warning message.

The full form of argument is:

```
action:message:category:module:line
```

Here, *action* is as explained above but only applies to messages that match the remaining fields. Empty fields match all values; trailing empty fields may be omitted. The *message* field matches the start of the warning message printed; this match is case-insensitive. The *category* field matches the warning category. This must be a class name; the match tests whether the actual warning category of the message is a subclass of the specified warning category. The full class name must be given. The *module* field matches the (fully-qualified) module name; this match is case-sensitive. The *line* field matches the line number, where zero matches all line numbers and is thus equivalent to an omitted line number.

See also: `warnings` – the warnings module
PEP 230 – Warning framework
`PYTHONWARNINGS`

-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`;
- `-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.
- `-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 the `tracemalloc.start()` for more information.
- `-X showalloccount` to output the total count of allocated objects for each type when the program finishes. This only works when Python was built with `COUNT_ALLOCS` defined.

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

Changed in version 3.2: The `-X` option was added.

New in version 3.3: The `-X faulthandler` option.

New in version 3.4: The `-X showrefcount` and `-X tracemalloc` options.

New in version 3.6: The `-X showalloccount` option.

1.1.4. Options you shouldn't use

-J

Reserved for use by Jython.

1.2. Environment variables

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

PYTHONHOME

Change the location of the standard Python libraries. By default, the libraries are searched in *prefix*`/lib/python`*version* and *exec_prefix*`/lib/python`*version*, 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 Interface options. The search path can be manipulated from within a Python program as the variable sys.path.

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.

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.

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.

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.

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 OS X.

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.

PYTHONHASHSEED

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

New in version 3.2.3.

PYTHONIOENCODING

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

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

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

Changed in version 3.6: On Windows, the encoding specified by this variable is ignored for interactive console buffers unless `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`.

> **See also:** **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 Distutils installation paths for `python setup.py install --user`.

> **See also:** **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 Mac OS X.

PYTHONWARNINGS

This is equivalent to the `-W` option. If set to a comma separated string, it is equivalent to specifying `-W` multiple times.

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.

New in version 3.3.

PYTHONTRACEMALLOC

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

New in version 3.4.

PYTHONASYNCIODEBUG

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

New in version 3.4.

PYTHONMALLOC

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

Set the family of memory allocators used by Python:

- `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.

Install debug hooks:

- `debug`: install debug hooks on top of the default memory allocator
- `malloc_debug`: same as `malloc` but also install debug hooks
- `pymalloc_debug`: same as `pymalloc` but also install debug hooks

When Python is compiled in release mode, the default is `pymalloc`. When compiled in debug mode, the default is `pymalloc_debug` and the debug hooks are used automatically.

If Python is configured without `pymalloc` support, `pymalloc` and `pymalloc_debug` are not available, the default is `malloc` in release mode and `malloc_debug` in debug mode.

See the `PyMem_SetupDebugHooks()` function for debug hooks on Python memory allocators.

New in version 3.6.

PYTHONMALLOCSTATS

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

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

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

PYTHONLEGACYWINDOWSFSENCODING

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

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

Availability: Windows

New in version 3.6: See **PEP 529** for more details.

PYTHONLEGACYWINDOWSSTDIO

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

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

Availability: Windows

New in version 3.6.

1.2.1. Debug-mode variables

Setting these variables only has an effect in a debug build of Python, that is, if Python was configured with the `--with-pydebug` build option.

PYTHONTHREADDEBUG

If set, Python will print threading debug info.

PYTHONDUMPREFS

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