Interpreter Settings

sys contains attributes and functions for accessing compile-time or runtime configuration settings for the interpreter.

Build-time Version Information

The version used to build the C interpreter is available in a few forms. `sys.version` is a human-readable string that usually includes the full version number as well as information about the build date, compiler, and platform. `sys.hexversion` is easier to use for checking the interpreter version since it is a simple integer. When formatted using `hex()`, it is clear that parts of `sys.hexversion` come from the version information also visible in the more readable `sys.version_info` (a five-part namedtuple representing just the version number). The separate C API version used by the current interpreter is saved in `sys.api_version`.

```
# sys_version_values.py

import sys

print('Version info:')
print()
print('sys.version      =', repr(sys.version))
print('sys.version_info =', sys.version_info)
print('sys.hexversion   =', hex(sys.hexversion))
print('sys.api_version  =', sys.api_version)
```

All of the values depend on the actual interpreter used to run the sample program.

```
$ python3 sys_version_values.py

Version info:

sys.version      = '3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018,
03:13:28) \n[Clang 6.0 (clang-600.0.57)]'
sys.version_info = sys.version_info(major=3, minor=7, micro=1,
releaselevel='final', serial=0)
sys.hexversion   = 0x30701f0
sys.api_version  = 1013
```

The operating system platform used to build the interpreter is saved as `sys.platform`.

```
# sys_platform.py

import sys

print('This interpreter was built for:', sys.platform)
```

For most Unix systems, the value comes from combining the output of `uname -s` with the first part of the version in `uname -r`. For other operating systems there is a hard-coded table of values.

```
$ python3 sys_platform.py

This interpreter was built for: darwin
```

See also

- Platform values – Hard-coded values of `sys.platform` for systems without `uname`.

Interpreter Implementation

The CPython interpreter is one of several implementations of the Python language. `sys.implementation` is provided to detect the current implementation for libraries that need to work around any differences in interpreters.

```
# sys_implementation.py

import sys


print('Name:', sys.implementation.name)
print('Version:', sys.implementation.version)
print('Cache tag:', sys.implementation.cache_tag)
```

`sys.implementation.version` is the same as `sys.version_info` for CPython, but will be different for other interpreters.

```
$ python3 sys_implementation.py

Name: cpython
Version: sys.version_info(major=3, minor=7, micro=1, releaseleve
l='final', serial=0)
Cache tag: cpython-37
```

See also

- **PEP 421** – Adding `sys.implementation`

Command Line Options

The CPython interpreter accepts several command-line options to control its behavior, listed in the table below.

CPython Command Line Option Flags

Option	Meaning
`-B`	do not write .py[co] files on import
`-b`	issue warnings about converting bytes to string without decoding properly and comparing bytes with strings
`-bb`	convert bytes warnings to errors
`-d`	debug output from parser
`-E`	ignore PYTHON* environment variables (such as PYTHONPATH)
`-i`	inspect interactively after running script
`-O`	optimize generated bytecode slightly
`-OO`	remove doc-strings in addition to the -O optimizations
`-s`	do not add user site directory to sys.path
`-S`	do not run ‘import site’ on initialization
`-t`	issue warnings about inconsistent tab usage
`-tt`	issue errors for inconsistent tab usage
`-v`	verbose

Some of these are available for programs to check through `sys.flags`.

```
# sys_flags.py
```

```
import sys

if sys.flags.bytes_warning:
    print('Warning on bytes/str errors')
if sys.flags.debug:
    print('Debuging')
if sys.flags.inspect:
    print('Will enter interactive mode after running')
if sys.flags.optimize:
    print('Optimizing byte-code')
```

```
    print('Optimizing byte-code')
if sys.flags.dont_write_bytecode:
    print('Not writing byte-code files')
if sys.flags.no_site:
    print('Not importing "site"')
if sys.flags.ignore_environment:
    print('Ignoring environment')
if sys.flags.verbose:
    print('Verbose mode')
```

Experiment with `sys_flags.py` to learn how the command line options map to the flags settings.

```
$ python3 -S -E -b sys_flags.py

Warning on bytes/str errors
Not importing "site"
Ignoring environment
```

Unicode Defaults

To get the name of the default Unicode encoding the interpreter is using, call `getdefaultencoding()`. The value is set during start-up, and cannot be changed.

The internal encoding default and the file system encoding may be different for some operating systems, so there is a separate way to retrieve the file system setting. `getfilesystemencoding()` returns an OS-specific (*not* file system-specific) value.

```
# sys_unicode.py

import sys

print('Default encoding     :', sys.getdefaultencoding())
print('File system encoding :', sys.getfilesystemencoding())
```

Rather than relying on the global default encoding, most Unicode experts recommend making an application explicitly Unicode-aware. This provides two benefits: different Unicode encodings for different data sources can be handled more cleanly, and the number of assumptions about encodings in the application code is reduced.

```
$ python3 sys_unicode.py

Default encoding     : utf-8
File system encoding : utf-8
```

Interactive Prompts

The interactive interpreter uses two separate prompts for indicating the default input level (`ps1`) and the “continuation” of a multi-line statement (`ps2`). The values are only used by the interactive interpreter.

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>>
```

Either or both prompt can be changed to a different string.

```
>>> sys.ps1 = '::: '
::: sys.ps2 = '~~~ '
::: for i in range(3):
~~~   print i
~~~
0
1
2
:::
```

Alternately, any object that can be converted to a string (via `__str__`) can be used for the prompt.

Alternately, any object that can be converted to a string (via __str__) can be used for the prompt.

```
# sys_ps1.py

import sys


class LineCounter:

    def __init__(self):
        self.count = 0

    def __str__(self):
        self.count += 1
        return '({:3d})> '.format(self.count)
```

The `LineCounter` keeps track of how many times it has been used, so the number in the prompt increases each time.

```
$ python
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> from sys_ps1 import LineCounter
>>> import sys
>>> sys.ps1 = LineCounter()
(  1)>
(  2)>
(  3)>
```

Display Hook

`sys.displayhook` is invoked by the interactive interpreter each time the user enters an expression. The result of evaluating the expression is passed as the only argument to the function.

```
# sys_displayhook.py

import sys


class ExpressionCounter:

    def __init__(self):
        self.count = 0
        self.previous_value = self

    def __call__(self, value):
        print()
        print('  Previous:', self.previous_value)
        print('  New     :', value)
        print()
        if value != self.previous_value:
            self.count += 1
            sys.ps1 = '({:3d})> '.format(self.count)
        self.previous_value = value
        sys.__displayhook__(value)


print('installing')
sys.displayhook = ExpressionCounter()
```

The default value (saved in `sys.__displayhook__`) prints the result to stdout and saves it in _ for easy reference later.

```
$ python3
Python 3.4.2 (v3.4.2:ab2c023a9432, Oct  5 2014, 20:42:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> import sys_displayhook
```

```
installing
>>> 1 + 2

  Previous: <sys_displayhook.ExpressionCounter
  object at 0x1021035f8>
  New     : 3

3
(  1)> 'abc'

  Previous: 3
  New     : abc

'abc'
(  2)> 'abc'

  Previous: abc
  New     : abc

'abc'
(  2)> 'abc' * 3

  Previous: abc
  New     : abcabcabc

'abcabcabc'
(  3)>
```

Install Location

The path to the actual interpreter program is available in `sys.executable` on all systems for which having a path to the interpreter makes sense. This can be useful for ensuring that the correct interpreter is being used, and also gives clues about paths that might be set based on the interpreter location.

`sys.prefix` refers to the parent directory of the interpreter installation. It usually includes `bin` and `lib` directories for executables and installed modules, respectively.

```
# sys_locations.py

import sys

print('Interpreter executable:')
print(sys.executable)
print('\nInstallation prefix:')
print(sys.prefix)
```

This example output was produced on a Mac running a framework build installed from python.org.

```
$ python3 sys_locations.py

Interpreter executable:
/Library/Frameworks/Python.framework/Versions/3.5/bin/python3

Installation prefix:
/Library/Frameworks/Python.framework/Versions/3.5
```

sys — System-specific Configuration

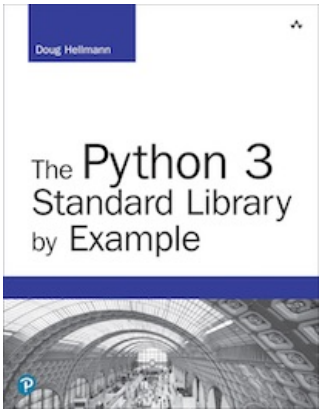
Runtime Environment

Quick Links

This page was last updated 2018-12-09.

Navigation

sys — System-specific Configuration
Runtime Environment



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example