28.5. `warnings` — Warning control¶

New in version 2.1.

Warning messages are typically issued in situations where it is useful to alert the user of some condition in a program, where that condition (normally) doesn't warrant raising an exception and terminating the program. For example, one might want to issue a warning when a program uses an obsolete module.

Python programmers issue warnings by calling the `warn()` function defined in this module. (C programmers use `PyErr_WarnEx()`; see *Exception Handling* for details).

Warning messages are normally written to `sys.stderr`, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions. The disposition of warnings can vary based on the warning category (see below), the text of the warning message, and the source location where it is issued. Repetitions of a particular warning for the same source location are typically suppressed.

There are two stages in warning control: first, each time a warning is issued, a determination is made whether a message should be issued or not; next, if a message is to be issued, it is formatted and printed using a user-settable hook.

The determination whether to issue a warning message is controlled by the warning filter, which is a sequence of matching rules and actions. Rules can be added to the filter by calling `filterwarnings()` and reset to its default state by calling `resetwarnings()`.

The printing of warning messages is done by calling `showwarning()`, which may be overridden; the default implementation of this function formats the message by calling `formatwarning()`, which is also available for use by custom implementations.

28.5.1. Warning Categories¶

There are a number of built-in exceptions that represent warning categories. This categorization is useful to be able to filter out groups of warnings. The following warnings category classes are currently defined:

Class	Description
`Warning`	This is the base class of all warning category classes. It is a subclass of `Exception`.
`UserWarning`	The default category for `warn()`.
`DeprecationWarning`	Base category for warnings about deprecated features.
`SyntaxWarning`	Base category for warnings about dubious syntactic features.
`RuntimeWarning`	Base category for warnings about dubious runtime features.
`FutureWarning`	Base category for warnings about constructs that will change semantically in the future.
`PendingDeprecationWarning`	Base category for warnings about features that will be deprecated in the future (ignored by default).
`ImportWarning`	Base category for warnings triggered during the process of importing a module (ignored by default).
`UnicodeWarning`	Base category for warnings related to Unicode.

While these are technically built-in exceptions, they are documented here, because conceptually they belong to the warnings mechanism.

User code can define additional warning categories by subclassing one of the standard warning categories. A warning category must always be a subclass of the `Warning` class.

28.5.2. The Warnings Filter¶

The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

Conceptually, the warnings filter maintains an ordered list of filter specifications; any specific warning is matched against each filter specification in the list in turn until a match is found; the match determines the disposition of the match. Each entry is a tuple of the form (*action*, *message*, *category*, *module*, *lineno*), where:

action is one of the following strings:

Value	Disposition
`"error"`	turn matching warnings into exceptions

`"ignore"`	never print matching warnings
`"always"`	always print matching warnings
`"default"`	print the first occurrence of matching warnings for each location where the warning is issued
`"module"`	print the first occurrence of matching warnings for each module where the warning is issued
`"once"`	print only the first occurrence of matching warnings, regardless of location

message is a string containing a regular expression that the warning message must match (the match is compiled to always be case-insensitive).

category is a class (a subclass of `Warning`) of which the warning category must be a subclass in order to match.

module is a string containing a regular expression that the module name must match (the match is compiled to be case-sensitive).

lineno is an integer that the line number where the warning occurred must match, or `0` to match all line numbers.

Since the `Warning` class is derived from the built-in `Exception` class, to turn a warning into an error we simply raise `category(message)`.

The warnings filter is initialized by *-W* options passed to the Python interpreter command line. The interpreter saves the arguments for all *-W* options without interpretation in `sys.warnoptions`; the `warnings` module parses these when it is first imported (invalid options are ignored, after printing a message to `sys.stderr`).

The warnings that are ignored by default may be enabled by passing *-Wd* to the interpreter. This enables default handling for all warnings, including those that are normally ignored by default. This is particular useful for enabling ImportWarning when debugging problems importing a developed package. ImportWarning can also be enabled explicitly in Python code using:

```
warnings.simplefilter('default', ImportWarning)
```

28.5.3. Temporarily Suppressing Warnings¶

If you are using code that you know will raise a warning, such as a deprecated function, but do not want to see the warning, then it is possible to suppress the warning using the `catch_warnings` context manager:

```
import warnings

def fxn():
   warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings():
   warnings.simplefilter("ignore")
   fxn()
```

While within the context manager all warnings will simply be ignored. This allows you to use known-deprecated code without having to see the warning while not suppressing the warning for other code that might not be aware of its use of deprecated code.

28.5.4. Testing Warnings¶

To test warnings raised by code, use the `catch_warnings` context manager. With it you can temporarily mutate the warnings filter to facilitate your testing. For instance, do the following to capture all raised warnings to check:

```
import warnings

def fxn():
   warnings.warn("deprecated", DeprecationWarning)

with warnings.catch_warnings(record=True) as w:
   # Cause all warnings to always be triggered.
   warnings.simplefilter("always")
   # Trigger a warning.
   fxn()
   # Verify some things
   assert len(w) == 1
   assert issubclass(w[-1].category, DeprecationWarning)
   assert "deprecated" in str(w[-1].message)
```

One can also cause all warnings to be exceptions by using `error` instead of `always`. One thing to be aware of is that if a warning has already been raised because of a `once`/`default` rule, then no matter what filters are set the warning will not be seen again unless the warnings registry related to the warning has been cleared.

Once the context manager exits, the warnings filter is restored to its state when the context was entered. This prevents tests from changing the warnings filter in unexpected ways between tests and leading to indeterminate test results. The `showwarning()` function in the module is also restored to its original value.

When testing multiple operations that raise the same kind of warning, it is important to test them in a manner that confirms each operation is raising a new warning (e.g. set warnings to be raised as exceptions and check the operations raise exceptions, check that the length of the warning list continues to increase after each operation, or else delete the previous entries from the warnings list before each new operation).

28.5.5. Available Functions¶

`warnings.warn`(*message*[, *category*[, *stacklevel*]])¶

Issue a warning, or maybe ignore it or raise an exception. The *category* argument, if given, must be a warning category class (see above); it defaults to `UserWarning`. Alternatively *message* can be a `Warning` instance, in which case *category* will be ignored and `message.__class__` will be used. In this case the message text will be `str(message)`. This function raises an exception if the particular warning issued is changed into an error by the warnings filter see above. The *stacklevel* argument can be used by wrapper functions written in Python, like this:

```
def deprecation(message):
   warnings.warn(message, DeprecationWarning, stacklevel=2)
```

This makes the warning refer to `deprecation()`‘s caller, rather than to the source of `deprecation()` itself (since the latter would defeat the purpose of the warning message).

`warnings.warn_explicit`(*message*, *category*, *filename*, *lineno*[, *module*[, *registry*[, *module_globals*]]])¶

This is a low-level interface to the functionality of `warn()`, passing in explicitly the message, category, filename and line number, and optionally the module name and the registry (which should be the `__warningregistry__` dictionary of the module). The module name defaults to the filename with `.py` stripped; if no registry is passed, the warning is never suppressed. *message* must be a string and *category* a subclass of `Warning` or *message* may be a `Warning` instance, in which case *category* will be ignored.

module_globals, if supplied, should be the global namespace in use by the code for which the warning is issued. (This argument is used to support displaying source for modules found in zipfiles or other non-filesystem import sources).

Changed in version 2.5: Added the *module_globals* parameter.

`warnings.warnpy3k`(*message*[, *category*[, *stacklevel*]])¶

Issue a warning related to Python 3.x deprecation. Warnings are only shown when Python is started with the -3 option. Like `warn()` *message* must be a string and *category* a subclass of `Warning`. `warnpy3k()` is using `DeprecationWarning` as default warning class.

New in version 2.6.

`warnings.showwarning`(*message*, *category*, *filename*, *lineno*[, *file*[, *line*]])¶

Write a warning to a file. The default implementation calls `formatwarning(message, category, filename, lineno, line)` and writes the resulting string to *file*, which defaults to `sys.stderr`. You may replace this function with an alternative implementation by assigning to `warnings.showwarning`. *line* is a line of source code to be included in the warning message; if *line* is not supplied, `showwarning()` will try to read the line specified by *filename* and *lineno*.

Changed in version 2.6: Added the *line* argument. Implementations that lack the new argument will trigger a `DeprecationWarning`.

`warnings.formatwarning`(*message*, *category*, *filename*, *lineno*[, *line*])¶

Format a warning the standard way. This returns a string which may contain embedded newlines and ends in a newline. *line* is a line of source code to be included in the warning message; if *line* is not supplied, `formatwarning()` will try to read the line specified by *filename* and *lineno*.

Changed in version 2.6: Added the *line* argument.

`warnings.filterwarnings`(*action*[, *message*[, *category*[, *module*[, *lineno*[, *append*]]]]])¶
Insert an entry into the list of *warnings filter specifications*. The entry is inserted at the front by default; if *append* is true, it is inserted at the end. This checks the types of the arguments, compiles the *message* and *module* regular expressions, and inserts them as a tuple in the list of warnings filters. Entries closer to the front of the list override entries later in the list, if both match a particular warning. Omitted arguments default to a value that matches everything.

`warnings.simplefilter`(*action*[, *category*[, *lineno*[, *append*]]])¶
Insert a simple entry into the list of *warnings filter specifications*. The meaning of the function parameters is as for `filterwarnings()`, but regular expressions are not needed as the filter inserted always matches any message in any module as long as the category and line number match.

`warnings.resetwarnings`()¶
Reset the warnings filter. This discards the effect of all previous calls to `filterwarnings()`, including that of the *-W* command line options and calls to `simplefilter()`.

28.5.6. Available Context Managers¶

class `warnings.catch_warnings`([*, *record=False*, *module=None*])¶

A context manager that copies and, upon exit, restores the warnings filter and the `showwarning()` function. If the *record* argument is `False` (the default) the context manager returns `None` on entry. If *record* is `True`, a list is returned that is progressively populated with objects as seen by a custom `showwarning()` function (which also suppresses output to `sys.stdout`). Each object in the list has attributes with the same names as the arguments to `showwarning()`.

The *module* argument takes a module that will be used instead of the module returned when you import `warnings` whose filter will be protected. This argument exists primarily for testing the `warnings` module itself.

Note

In Python 3.0, the arguments to the constructor for `catch_warnings` are keyword-only arguments.

New in version 2.6.

Table Of Contents

Previous topic

28.4. `__main__` — Top-level script environment

Next topic

28.6. `contextlib` — Utilities for `with`-statement contexts

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 28. Python Runtime Services »