# `!warnings` --- Warning control


- Warning Categories
- The Warnings Filter
  - Repeated Warning Suppression Criteria
  - Describing Warning Filters
  - Default Warning Filter
  - Overriding the default filter
- Temporarily Suppressing Warnings
- Testing Warnings
- Updating Code For New Versions of Dependencies
- Available Functions
- Available Context Managers
- Concurrent safety of Context Managers


warnings

**Source code:** `Lib/warnings.py`

single: warnings

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 `exceptionhandling` 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 <warning-categories>`, 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 <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.

`logging.captureWarnings` allows you to handle all warnings with the standard logging infrastructure.

# 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.

While these are technically `built-in exceptions <warning-categories-as-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.

The following warnings category classes are currently defined:

|l|p{0.6linewidth}|

| Class | Description |
|---|---|
| `Warning` | Base class for warning categories. It is a subclass of `Exception`. |
| `UserWarning` | Base class for warnings generated by user code. The default category for `warn`. |
| `DeprecationWarning` | Base class for warnings about deprecated features when those warnings are intended for other Python developers (ignored by default, unless triggered by code in `__main__`). |
| `PendingDeprecationWarning` | Base class for warnings about features that will be deprecated in the future (ignored by default). |

| Class | Description |
| --- | --- |
| `SyntaxWarning` | Base class for warnings about dubious syntax (typically emitted when compiling Python source code, and hence may not be suppressed by runtime filters). |
| `RuntimeWarning` | Base class for warnings about dubious runtime behavior. |
| `FutureWarning` | Base class for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python. |
| `ImportWarning` | Base class for warnings triggered during the process of importing a module (ignored by default). |
| `UnicodeWarning` | Base class for warnings related to Unicode. |
| `EncodingWarning` | Base class for warnings related to encodings. See `io-encoding-warning` for details. |
| `BytesWarning` | Base class for warnings related to `bytes` and `bytearray`. |
| `ResourceWarning` | Base class for warnings related to resource usage (ignored by default). |

3.7 Previously `DeprecationWarning` and `FutureWarning` were distinguished based on whether a feature was being removed entirely or changing its behaviour. They are now distinguished based on their intended audience and the way they're handled by the default warnings filters.

# 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 filter 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 |
| --- | --- |
| `"default"` | print the first occurrence of matching warnings for each location (module + line number) where the warning is issued |
| `"error"` | turn matching warnings into exceptions |
| `"ignore"` | never print matching warnings |
| `"always"` | always print matching warnings |

| Value | Disposition |
| --- | --- |
| `"all"` | alias to "always" |
| `"module"` | print the first occurrence of matching warnings for each module where the warning is issued (regardless of line number) |
| `"once"` | print only the first occurrence of matching warnings, regardless of location |

- *message* is a string containing a regular expression that the start of the warning message must match, case-insensitively. In `-W` and `PYTHONWARNINGS`, if *message* starts and ends with a forward slash (`/`), it specifies a regular expression as above; otherwise it is a literal string that the start of the warning message must match (case-insensitively), ignoring any whitespace at the start or end of *message*.
- *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 start of the fully qualified module name must match, case-sensitively. In `-W` and `PYTHONWARNINGS`, if *module* starts and ends with a forward slash (`/`), it specifies a regular expression as above; otherwise it is a literal string that the fully qualified module name must be equal to (case-sensitively), ignoring any whitespace at the start or end of *module*.
- *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)`.

If a warning is reported and doesn't match any registered filter then the "default" action is applied (hence its name).

# Repeated Warning Suppression Criteria

The filters that suppress repeated warnings apply the following criteria to determine if a warning is considered a repeat:

- `"default"`: A warning is considered a repeat only if the (*message*, *category*, *module*, *lineno*) are all the same.
- `"module"`: A warning is considered a repeat if the (*message*, *category*, *module*) are the same, ignoring the line number.
- `"once"`: A warning is considered a repeat if the (*message*, *category*) are the same, ignoring the module and line number.

# Describing Warning Filters

The warnings filter is initialized by `-W` options passed to the Python interpreter command line and the `PYTHONWARNINGS` environment variable. The interpreter saves the arguments for all supplied entries 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`).

Individual warnings filters are specified as a sequence of fields separated by colons:

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

The meaning of each of these fields is as described in `warning-filter`. When listing multiple filters on a single line (as for `PYTHONWARNINGS`), the individual filters are separated by commas and the filters listed later take precedence over those listed before them (as they're applied left-to-right, and the most recently applied filters take precedence over earlier ones).

Commonly used warning filters apply to either all warnings, warnings in a particular category, or warnings raised by particular modules or packages. Some examples:

```
default                      # Show all warnings (even those ignored by default)
ignore                       # Ignore all warnings
error                        # Convert all warnings to errors
error::ResourceWarning       # Treat ResourceWarning messages as errors
default::DeprecationWarning  # Show DeprecationWarning messages
ignore,default:::mymodule    # Only report warnings triggered by "mymodule"
error:::mymodule             # Convert warnings to errors in "mymodule"
```

# Default Warning Filter

By default, Python installs several warning filters, which can be overridden by the `-W` command-line option, the `PYTHONWARNINGS` environment variable and calls to `filterwarnings`.

In regular release builds, the default warning filter has the following entries (in order of precedence):

```
default::DeprecationWarning:__main__
ignore::DeprecationWarning
ignore::PendingDeprecationWarning
ignore::ImportWarning
ignore::ResourceWarning
```

In a `debug build <debug-build>`, the list of default warning filters is empty.

3.2 `DeprecationWarning` is now ignored by default in addition to `PendingDeprecationWarning`.

3.7 `DeprecationWarning` is once again shown by default when triggered directly by code in `__main__`.

3.7 `BytesWarning` no longer appears in the default filter list and is instead configured via `sys.warnoptions` when `-b` is specified twice.

# Overriding the default filter

Developers of applications written in Python may wish to hide *all* Python level warnings from their users by default, and only display them when running tests or otherwise working on the application. The `sys.warnoptions` attribute used to pass filter configurations to the interpreter can be used as a marker to indicate whether or not warnings should be disabled:

```
import sys

if not sys.warnoptions:
    import warnings
    warnings.simplefilter("ignore")
```

Developers of test runners for Python code are advised to instead ensure that *all* warnings are displayed by default for the code under test, using code like:

```
import sys

if not sys.warnoptions:
    import os, warnings
    warnings.simplefilter("default") # Change the filter in this process
    os.environ["PYTHONWARNINGS"] = "default" # Also affect subprocesses
```

Finally, developers of interactive shells that run user code in a namespace other than `__main__` are advised to ensure that `DeprecationWarning` messages are made visible by default, using code like the following (where `user_ns` is the module used to execute code entered interactively):

```
import warnings
warnings.filterwarnings("default", category=DeprecationWarning,
                                   module=user_ns.get("__name__"))
```

# 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 (even when warnings have been explicitly configured via the command line), 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.

> “
>
> > “[!NOTE] See `warning-concurrent-safe` for details on the concurrency-safety of the `catch_warnings` context manager when used in programs using multiple threads or async functions.

# 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.

> “
>
> > “ [!NOTE] See `warning-concurrent-safe` for details on the concurrency-safety of the `catch_warnings` context manager when used in programs using multiple threads or async functions.

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).

# Updating Code For New Versions of Dependencies

Warning categories that are primarily of interest to Python developers (rather than end users of applications written in Python) are ignored by default.

Notably, this "ignored by default" list includes `DeprecationWarning` (for every module except `__main__`), which means developers should make sure to test their code with typically ignored warnings made visible in order to receive timely notifications of future breaking API changes (whether in the standard library or third party packages).

In the ideal case, the code will have a suitable test suite, and the test runner will take care of implicitly enabling all warnings when running tests (the test runner provided by the `unittest` module does this).

In less ideal cases, applications can be checked for use of deprecated interfaces by passing `-Wd <-W>` to the Python interpreter (this is shorthand for `!-W default`) or setting `PYTHONWARNINGS=default` in the environment. This enables default handling for all warnings, including those that are ignored by default. To change what action is taken for encountered warnings you can change what argument is passed to `-W` (e.g. `!-W error`). See the `-W` flag for more details on what is possible.

# Available Functions

warn(message, category=None, stacklevel=1, source=None, *, skip_file_prefixes=())

Issue a warning, or maybe ignore it or raise an exception. The *category* argument, if given, must be a `warning category class <warning-categories>`; 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 <warning-filter>`. The *stacklevel* argument can be used by wrapper functions written in Python, like this:

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

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

The *skip_file_prefixes* keyword argument can be used to indicate which stack frames are ignored when counting stack levels. This can be useful when you want the warning to always appear at call sites outside of a package when a constant *stacklevel* does not fit all call paths or is otherwise challenging to maintain. If supplied, it must be a tuple of strings. When prefixes are supplied, stacklevel is implicitly overridden to be `max(2, stacklevel)`. To cause a warning to be attributed to the caller from outside of the current package you might write:

```
# example/lower.py
_warn_skips = (os.path.dirname(__file__),)

def one_way(r_luxury_yacht=None, t_wobbler_mangrove=None):
    if r_luxury_yacht:
        warnings.warn("Please migrate to t_wobbler_mangrove=.",
                      skip_file_prefixes=_warn_skips)
```

```
# example/higher.py
from . import lower

def another_way(**kw):
    lower.one_way(**kw)
```

This makes the warning refer to both the `example.lower.one_way()` and `example.higher.another_way()` call sites only from calling code living outside of `example` package.

*source*, if supplied, is the destroyed object which emitted a `ResourceWarning`.

3.6 Added *source* parameter.

3.12 Added *skip_file_prefixes*.

warn_explicit(message, category, filename, lineno, module=None, registry=None, module_globals=None, source=None)

This is a low-level interface to the functionality of `warn`, passing in explicitly the message, category, filename and line number, and optionally other arguments. *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*, if supplied, should be the module name. If no module is passed, the module regular expression in `warnings filter <warning-filter>` will be tested against the module names constructed from the path components starting from all parent directories (with `/__init__.py`, `.py` and, on Windows, `.pyw` stripped) and against the filename with `.py` stripped. For example, when the filename is `'/path/to/package/module.py'`, it will be tested against `'path.to.package.module'`, `'to.package.module'` `'package.module'`, `'module'`, and `'/path/to/package/module'`.

*registry*, if supplied, should be the `__warningregistry__` dictionary of the module. If no registry is passed, each warning is treated as the first occurrence, that is, filter actions `"default"`, `"module"` and `"once"` are handled as `"always"`.

*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).

*source*, if supplied, is the destroyed object which emitted a `ResourceWarning`.

3.6 Add the *source* parameter.

3.15 If no module is passed, test the filter regular expression against module names created from the path, not only the path itself.

showwarning(message, category, filename, lineno, file=None, line=None)

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 any callable 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*.

formatwarning(message, category, filename, lineno, line=None)

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*.

filterwarnings(action, message='', category=Warning, module='', lineno=0, append=False)

Insert an entry into the list of `warnings filter specifications <warning-filter>`. 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.

simplefilter(action, category=Warning, lineno=0, append=False)

Insert a simple entry into the list of `warnings filter specifications <warning-filter>`. 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.

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`.

deprecated(message, /, *, category=DeprecationWarning, stacklevel=1)

Decorator to indicate that a class, function or overload is deprecated.

When this decorator is applied to an object, deprecation warnings may be emitted at runtime when the object is used. `static type checkers <static type checker>` will also generate a diagnostic on usage of the deprecated object.

Usage:

```
from warnings import deprecated
from typing import overload

@deprecated("Use B instead")
class A:
    pass

@deprecated("Use g instead")
def f():
    pass

@overload
@deprecated("int support is deprecated")
def g(x: int) -> int: ...
@overload
def g(x: str) -> int: ...
```

The warning specified by *category* will be emitted at runtime on use of deprecated objects. For functions, that happens on calls; for classes, on instantiation and on creation of subclasses. If the *category* is `None`, no warning is emitted at runtime. The *stacklevel* determines where the warning is emitted. If it is `1` (the default), the warning is emitted at the direct caller of the deprecated object; if it is higher, it is emitted further up the stack. Static type checker behavior is not affected by the *category* and *stacklevel* arguments.

The deprecation message passed to the decorator is saved in the `__deprecated__` attribute on the decorated object. If applied to an overload, the decorator must be after the `~typing.overload` decorator for the attribute to exist on the overload as returned by `typing.get_overloads`.

3.13 See `702`.

# Available Context Managers

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.stderr`). Each object in the list is guaranteed to have the following attributes:

> - `message`: the warning message (an instance of `Warning`)
> - `category`: the warning category (a subclass of `Warning`)
> - `filename`: the file name where the warning occurred (`str`)
> - `lineno`: the line number in the file (`int`)
> - `file`: the file object used for output (if any), or `None`
> - `line`: the line of source code (if available), or `None`
> - `source`: the original object that generated the warning (if available), or `None`
> - `module`: the module name where the warning occurred (`str`), or `None`

3.6 The `source` attribute was added.

3.15 The `module` attribute was added.

The type of these objects is not specified and may change; only the presence of these attributes is guaranteed.

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.

If the *action* argument is not `None`, the remaining arguments are passed to `simplefilter` as if it were called immediately on entering the context.

See `warning-filter` for the meaning of the *category* and *lineno* parameters.

> [!NOTE] See `warning-concurrent-safe` for details on the concurrency-safety of the `catch_warnings` context manager when used in programs using multiple threads or async functions.

3.11

Added the *action*, *category*, *lineno*, and *append* parameters.

# Concurrent safety of Context Managers

The behavior of `catch_warnings` context manager depends on the `sys.flags.context_aware_warnings` flag. If the flag is true, the context manager behaves in a concurrent-safe fashion and otherwise not. Concurrent-safe means that it is both thread-safe and safe to use within `asyncio coroutines <coroutine>` and tasks. Being thread-safe means that behavior is predictable in a multi-threaded program. The flag defaults to true for free-threaded builds and false otherwise.

If the `~sys.flags.context_aware_warnings` flag is false, then `catch_warnings` will modify the global attributes of the `!warnings` module. This is not safe if used within a concurrent program (using multiple threads or using asyncio coroutines). For example, if two or more threads use the `catch_warnings` class at the same time, the behavior is undefined.

If the flag is true, `catch_warnings` will not modify global attributes and will instead use a `~contextvars.ContextVar` to store the newly established warning filtering state. A context variable provides thread-local storage and it makes the use of `catch_warnings` thread-safe.

The *record* parameter of the context handler also behaves differently depending on the value of the flag. When *record* is true and the flag is false, the context manager works by replacing and then later restoring the module's `showwarning` function. That is not concurrent-safe.

When *record* is true and the flag is true, the `showwarning` function is not replaced. Instead, the recording status is indicated by an internal property in the context variable. In this case, the `showwarning` function will not be restored when exiting the context handler.

The `~sys.flags.context_aware_warnings` flag can be set the `-X context_aware_warnings<-X>` command-line option or by the `PYTHON_CONTEXT_AWARE_WARNINGS` environment variable.

> “
>
> > “[!NOTE] It is likely that most programs that desire thread-safe behaviour of the warnings module will also want to set the `~sys.flags.thread_inherit_context` flag to true. That flag causes threads created by `threading.Thread` to start with a copy of the context variables from the thread starting it. When true, the context established by `catch_warnings` in one thread will also apply to new threads started by it. If false, new threads will start with an empty warnings context variable, meaning that any filtering that was established by a `catch_warnings` context manager will no longer be active.

3.14

Added the `sys.flags.context_aware_warnings` flag and the use of a context variable for `catch_warnings` if the flag is true. Previous versions of Python acted as if the flag was always set to false.

□□□ #2
□ □□□ □□□ 2026-10-31 17:23:51 UTC
□ □□□ □□□ 2026-10-31 17:23:53 UTC