Deprecated Documentation

Release 1.2.14

Marcos CARDOSO and Laurent LAPORTE

Jul 09, 2023

Contents

Welcome to Deprecated's Documentation. This documentation is divided into different parts. I recommend that you get started with *Installation* and then head over to the *Tutorial*. If you'd rather dive into the internals of the Deprecated Library, check out the *API* documentation.

CHAPTER 1

User's Guide

This part of the documentation, which is mostly prose, begins with some background information about Deprecated, then focuses on step-by-step instructions for using Deprecated.

1.1 Installation

1.1.1 Python Version

We recommend using the latest version of Python 3. Deprecated Library supports Python 3.4 and newer, Python 2.7 and newer, and PyPy 2.7 and 3.6.

1.1.2 Dependencies

This library uses the Wrapt library as a basis to construct function wrappers and decorator functions.

Development dependencies

These distributions will not be installed automatically. You need to install them explicitly with *pip install -e .[dev]*.

- pytest is a framework which makes it easy to write small tests, yet scales to support complex functional testing for applications and libraries…
- pytest-cov is a pytest plugin used to produce coverage reports.
- tox aims to automate and standardize testing in Python. It is part of a larger vision of easing the packaging, testing and release process of Python software…
- bump2version is a small command line tool to simplify releasing software by updating all version strings in your source code by the correct increment. Also creates commits and tags…
- sphinx is a tool that makes it easy to create intelligent and beautiful documentation.

1.1.3 Virtual environments

Use a virtual environment to manage the dependencies for your project, both in development and in production.

What problem does a virtual environment solve? The more Python projects you have, the more likely it is that you need to work with different versions of Python libraries, or even Python itself. Newer versions of libraries for one project can break compatibility in another project.

Virtual environments are independent groups of Python libraries, one for each project. Packages installed for one project will not affect other projects or the operating system's packages.

Python 3 comes bundled with the `venv` module to create virtual environments. If you're using a modern version of Python, you can continue on to the next section.

If you're using Python 2, see *Install virtualenv* first.

Create an environment

Create a project folder and a `venv` folder within:

```
mkdir myproject
cd myproject
python3 -m venv venv
```

On Windows:

```
py -3 -m venv venv
```

If you needed to install virtualenv because you are on an older version of Python, use the following command instead:

```
virtualenv venv
```

On Windows:

```
\Python27\Scripts\virtualenv.exe venv
```

Activate the environment

Before you work on your project, activate the corresponding environment:

```
. venv/bin/activate
```

On Windows:

```
venv\Scripts\activate
```

Your shell prompt will change to show the name of the activated environment.

1.1.4 Install Deprecated

Within the activated environment, use the following command to install Deprecated:

```
pip install Deprecated
```

Living on the edge

If you want to work with the latest Deprecated code before it's released, install or update the code from the master branch:

```
pip install -U https://github.com/tantale/deprecated/archive/master.tar.gz
```

1.1.5 Install virtualenv

If you are using Python 2, the venv module is not available. Instead, install virtualenv.

On Linux, virtualenv is provided by your package manager:

```
# Debian, Ubuntu
sudo apt-get install python-virtualenv

# CentOS, Fedora
sudo yum install python-virtualenv

# Arch
sudo pacman -S python-virtualenv
```

If you are on Mac OS X or Windows, download get-pip.py, then:

```
sudo python2 Downloads/get-pip.py
sudo python2 -m pip install virtualenv
```

On Windows, as an administrator:

```
\Python27\python.exe Downloads\get-pip.py
\Python27\python.exe -m pip install virtualenv
```

Now you can continue to *Create an environment*.

1.2 Introduction

1.2.1 What "Deprecated" Means

A function or class is deprecated when it is considered as it is no longer important. It is so unimportant, in fact, that you should no longer use it, since it has been superseded and may cease to exist in the future.

As a module, a class or function evolves, its API (Application Programming Interface) inevitably changes: functions are renamed for consistency, new and better methods are added, and attributes change. But such changes introduce a problem. You need to keep the old API around until developers make the transition to the new one, but you don't want them to continue programming to the old API.

The ability to deprecate a class or a function solves the problem. Python Standard Library does not provide a way to express deprecation easily. The Python Deprecated Library is here to fulfill this lack.

1.2.2 When to Deprecate

When you design an API, carefully consider whether it supersedes an old API. If it does, and you wish to encourage developers (users of the API) to migrate to the new API, then deprecate the old API. Valid reasons to deprecate an API

include:

- It is insecure, buggy, or highly inefficient;
- It is going away in a future release,
- It encourages bad coding practices.

Deprecation is a reasonable choice in all these cases because it preserves "backward compatibility" while encouraging developers to change to the new API. Also, the deprecation comments help developers decide when to move to the new API, and so should briefly mention the technical reasons for deprecation.

1.2.3 How to Deprecate

The Python Deprecated Library provides a `@deprecated` decorator to deprecate a class, method or function.

Using the decorator causes the Python interpreter to emit a warning at runtime, when an class instance is constructed, or a function is called. The warning is emitted using the Python warning control. Warning messages are normally written to `sys.stderr`, but their disposition can be changed flexibly, from ignoring all warnings to turning them into exceptions.

You are strongly recommended to use the `@deprecated` decorator with appropriate comments explaining how to use the new API. This ensures developers will have a workable migration path from the old API to the new API.

Example:

```
from deprecated import deprecated


@deprecated(version='1.2.0', reason="You should use another function")
def some_old_function(x, y):
    return x + y


class SomeClass(object):
    @deprecated(version='1.3.0', reason="This method is deprecated")
    def some_old_method(self, x, y):
        return x + y


some_old_function(12, 34)
obj = SomeClass()
obj.some_old_method(5, 8)
```

When you run this Python script, you will get something like this in the console:

```
$ pip install Deprecated
$ python hello.py
hello.py:15: DeprecationWarning: Call to deprecated function (or staticmethod) some_
↪old_function. (You should use another function) -- Deprecated since version 1.2.0.
  some_old_function(12, 34)
hello.py:17: DeprecationWarning: Call to deprecated method some_old_method. (This␣
↪method is deprecated) -- Deprecated since version 1.3.0.
  obj.some_old_method(5, 8)
```

1.3 Tutorial

In this tutorial, we will use the Deprecated Library to mark pieces of codes as deprecated. We will also see what's happened when a user tries to call deprecated codes.

1.3.1 Deprecated function

First, we have this little library composed of a single module: `liberty.py`:

```
# coding: utf-8
""" Liberty library is free """

import pprint


def print_value(value):
    """
    Print the value

    :param value: The value to print
    """
    pprint.pprint(value)
```

You decided to write a more powerful function called `better_print()` which will become a replacement of `print_value()`. And you decided that the later function is deprecated.

To mark the `print_value()` as deprecated, you can use the *`deprecated()`* decorator:

```
# coding: utf-8
""" Liberty library is free """

import pprint

from deprecated import deprecated


@deprecated
def print_value(value):
    """
    Print the value

    :param value: The value to print
    """
    pprint.pprint(value)


def better_print(value, printer=None):
    """
    Print the value using a *printer*.

    :param value: The value to print
    :param printer: Callable used to print the value, by default: :func:`pprint.
↪pprint`
    """
    printer = printer or pprint.pprint
    printer(value)
```

If the user tries to use the deprecated functions, he will have a warning for each call:

```
# coding: utf-8
import liberty

liberty.print_value("hello")
liberty.print_value("hello again")
liberty.better_print("Hi Tom!")
```

```
$ python use_liberty.py

using_liberty.py:4: DeprecationWarning: Call to deprecated function (or staticmethod)␣
↪print_value.
  liberty.print_value("hello")
'hello'
using_liberty.py:5: DeprecationWarning: Call to deprecated function (or staticmethod)␣
↪print_value.
  liberty.print_value("hello again")
'hello again'
'Hi Tom!'
```

As you can see, the deprecation warning is displayed like a stack trace. You have the source code path, the line number and the called function. This is very useful for debugging. But, this doesn't help the developer to choose a alternative: which function could he use instead?

To help the developer, you can add a "reason" message. For instance:

```
# coding: utf-8
""" Liberty library is free """

import pprint

from deprecated import deprecated


@deprecated("This function is rotten, use 'better_print' instead")
def print_value(value):
    """
    Print the value

    :param value: The value to print
    """
    pprint.pprint(value)


def better_print(value, printer=None):
    """
    Print the value using a *printer*.

    :param value: The value to print
    :param printer: Callable used to print the value, by default: :func:`pprint.
↪pprint`
    """
    printer = printer or pprint.pprint
    printer(value)
```

When the user calls the deprecated functions, he will have a more useful message:

```
$ python use_liberty.py

using_liberty.py:4: DeprecationWarning: Call to deprecated function (or staticmethod)␣
↪print_value. (This function is rotten, use 'better_print' instead)
  liberty.print_value("hello")
'hello'
using_liberty.py:5: DeprecationWarning: Call to deprecated function (or staticmethod)␣
↪print_value. (This function is rotten, use 'better_print' instead)
  liberty.print_value("hello again")
'hello again'
'Hi Tom!'
```

1.3.2 Deprecated method

Decorating a deprecated method works like decorating a function.

```
# coding: utf-8
""" Liberty library is free """

import pprint

from deprecated import deprecated


class Liberty(object):
    def __init__(self, value):
        self.value = value

    @deprecated("This method is rotten, use 'better_print' instead")
    def print_value(self):
        """ Print the value """
        pprint.pprint(self.value)

    def better_print(self, printer=None):
        """
        Print the value using a *printer*.

        :param printer: Callable used to print the value, by default: :func:`pprint.
↪pprint`
        """
        printer = printer or pprint.pprint
        printer(self.value)
```

When the user calls the deprecated methods, like this:

```
# coding: utf-8
import liberty

obj = liberty.Liberty("Greeting")
obj.print_value()
obj.print_value()
obj.better_print()
```

He will have:

```
$ python use_liberty.py

using_liberty.py:5: DeprecationWarning: Call to deprecated method print_value. (This␣
↪method is rotten, use 'better_print' instead)
  obj.print_value()
'Greeting'
using_liberty.py:6: DeprecationWarning: Call to deprecated method print_value. (This␣
↪method is rotten, use 'better_print' instead)
  obj.print_value()
'Greeting'
'Greeting'
```

Note: The call is done to the same object, so we have 3 times 'Greeting'.

1.3.3 Deprecated class

You can also decide that a class is deprecated.

For instance:

```
# coding: utf-8
""" Liberty library is free """

import pprint

from deprecated import deprecated


@deprecated("This class is not perfect")
class Liberty(object):
    def __init__(self, value):
        self.value = value

    def print_value(self):
        """ Print the value """
        pprint.pprint(self.value)
```

When the user use the deprecated class like this:

```
# coding: utf-8
import liberty

obj = liberty.Liberty("Salutation")
obj.print_value()
obj.print_value()
```

He will have a warning at object instantiation. Once the object is initialised, no more warning are emitted.

```
$ python use_liberty.py

using_liberty.py:4: DeprecationWarning: Call to deprecated class Liberty. (This class␣
↪is not perfect)
  obj = liberty.Liberty("Salutation")
'Salutation'
'Salutation'
```

If a deprecated class is used, then a warning message is emitted during class instantiation. In other word, deprecating a class is the same as deprecating it's `__new__` class method.

As a reminder, the magic method `__new__` will be called when instance is being created. Using this method you can customize the instance creation. the `deprecated()` decorator patches the `__new__` method in order to emmit the warning message before instance creation.

1.3.4 Controlling warnings

Warnings are emitted using the Python warning control. By default, Python installs several warning filters, which can be overridden by the -W command-line option, the PYTHONWARNINGS environment variable and calls to `warnings.filterwarnings()`. The warnings filter controls whether warnings are ignored, displayed, or turned into errors (raising an exception).

For instance:

```
import warnings
from deprecated import deprecated


@deprecated(version='1.2.1', reason="deprecated function")
def fun():
    print("fun")


if __name__ == '__main__':
    warnings.simplefilter("ignore", category=DeprecationWarning)
    fun()
```

When the user runs this script, the deprecation warnings are ignored in the main program, so no warning message are emitted:

```
$ python filter_warnings_demo.py

fun
```

1.3.5 Deprecation warning classes

The *`deprecated.classic.deprecated()`* and *`deprecated.sphinx.deprecated()`* functions are using the `DeprecationWarning` category but you can customize them by using your own category (or hierarchy of categories).

- *category* classes which you can use (among other) are:

Class	Description
`DeprecationWarning`	Base category for warnings about deprecated features when those warnings are intended for other Python developers (ignored by default, unless triggered by code in `__main__`).
`FutureWarning`	Base category for warnings about deprecated features when those warnings are intended for end users of applications that are written in Python.
`PendingDeprecationWarning`	Base category for warnings about features that will be deprecated in the future (ignored by default).

You can define your own deprecation warning hierarchy based on the standard deprecation classes.

For instance:

```
import warnings

from deprecated import deprecated


class MyDeprecationWarning(DeprecationWarning):
    """ My DeprecationWarning """


class DeprecatedIn26(MyDeprecationWarning):
    """ deprecated in 2.6 """


class DeprecatedIn30(MyDeprecationWarning):
    """ deprecated in 3.0 """


@deprecated(category=DeprecatedIn26, reason="deprecated function")
def foo():
    print("foo")


@deprecated(category=DeprecatedIn30, reason="deprecated function")
def bar():
    print("bar")


if __name__ == '__main__':
    warnings.filterwarnings("ignore", category=DeprecatedIn30)
    foo()
    bar()
```

When the user runs this script, the deprecation warnings for the 3.0 version are ignored:

```
$ python warning_classes_demo.py

 foo
 bar
 warning_classes_demo.py:30: DeprecatedIn26: Call to deprecated function (or␣
↪staticmethod) foo. (deprecated function)
   foo()
```

1.3.6 Filtering warnings locally

The *deprecated.classic.deprecated()* and *deprecated.sphinx.deprecated()* functions can change the warning filtering locally (at function calls).

- *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
`"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

You can define the *action* keyword parameter to override the filtering warnings locally.

For instance:

```
import warnings
from deprecated import deprecated


@deprecated(reason="do not call it", action="error")
def foo():
    print("foo")


if __name__ == '__main__':
    warnings.simplefilter("ignore")
    foo()
```

In this example, even if the global filter is set to "ignore", a call to the `foo()` function will raise an exception because the *action* is set to "error".

```
$ python filter_action_demo.py

Traceback (most recent call last):
  File "filter_action_demo.py", line 12, in <module>
    foo()
  File "path/to/deprecated/classic.py", line 274, in wrapper_function
    warnings.warn(msg, category=category, stacklevel=_stacklevel)
DeprecationWarning: Call to deprecated function (or staticmethod) foo. (do not call␣
↪it)
```

1.3.7 Modifying the deprecated code reference

By default, when a deprecated function or class is called, the warning message indicates the location of the caller.

The `extra_stacklevel` parameter allows customizing the stack level reference in the deprecation warning message.

This parameter is particularly useful in scenarios where you have a factory or utility function that creates deprecated objects or performs deprecated operations. By specifying an `extra_stacklevel` value, you can control the stack level at which the deprecation warning is emitted, making it appear as if the calling function is the deprecated one, rather than the actual deprecated entity.

For example, if you have a factory function `create_object()` that creates deprecated objects, you can use the `extra_stacklevel` parameter to emit the deprecation warning at the calling location. This provides clearer and more actionable deprecation messages, allowing developers to identify and update the code that invokes the deprecated functionality.

For instance:

```
import warnings

from deprecated import deprecated


@deprecated(version='1.0', extra_stacklevel=1)
class MyObject(object):
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return "object: {name}".format(name=self.name)


def create_object(name):
    return MyObject(name)


if __name__ == '__main__':
    warnings.filterwarnings("default", category=DeprecationWarning)
    # warn here:
    print(create_object("orange"))
    # and also here:
    print(create_object("banane"))
```

Please note that the `extra_stacklevel` value should be an integer indicating the number of stack levels to skip when emitting the deprecation warning.

1.4 The “Sphinx” decorators

1.4.1 Overview

Developers use the Deprecated library to decorate deprecated functions or classes. This is very practical, but you know that this library does more: you can also document your source code! How? It’s very simple: instead of using the “classic” decorator, you can use one of the “Sphinx” decorators.

The “Sphinx” decorators have the same function as the “classic” decorator but also allow you to add Sphinx directives in your functions or classes documentation (inside the docstring).

Attention: In Python 3.3 and previous versions, the docstring of a class is immutable[1], this means that you cannot use the “Sphinx” decorators. Naturally, this limitation does not exist in Python 3.4 and above.

1.4.2 What is a Sphinx directive?

Sphinx is a tool that makes it easy to create intelligent and beautiful documentation. This tool uses the reStructuredText (or Markdown) syntax to generate the documentation in different formats, the most common being HTML. Developers generally use this syntax to document the source code of their applications.

[1] See Issue 12773: classes should have mutable docstrings.

Sphinx offers several directives allowing to introduce a text block with a predefined role. Among all the directives, the ones that interest us are those related to the functions (or classes) life cycle, namely: `versionadded`, `versionchanged` and `deprecated`.

In the following example, the *mean()* function can be documented as follows:

```
def mean(values):
    """
    Compute the arithmetic mean ("average") of values.

    :type  values: typing.List[float]
    :param values: List of floats
    :return: Mean of values.

    .. deprecated:: 2.5.0
       Since Python 3.4, you can use the standard function :func:`statistics.mean`.
    """
    return sum(values) / len(values)
```

Therefore, the "Sphinx" decorators allow you to add a Sphinx directive to your functions or classes documentation. In the case of the `deprecated` directive, it obviously allows you to emit a `DeprecationWarning` warning.

1.4.3 Using the "Sphinx" decorators

The previous example can be writen using a "Sphinx" decorator:

```
from deprecated.sphinx import deprecated


@deprecated(
    reason="""Since Python 3.4, you can use the standard function :func:`statistics.
↪mean`.""",
    version="2.5.0",
)
def mean(values):
    """
    Compute the arithmetic mean ("average") of values.

    :type  values: typing.List[float]
    :param values: List of floats
    :return: Mean of values.
    """
    return sum(values) / len(values)
```

You can see the generated documentation with this simple call:

```
from calc_mean_deco import mean

print(mean.__doc__)
```

The documentation of the *mean()* function looks like this:

```
Compute the arithmetic mean ("average") of values.

:type  values: typing.List[float]
:param values: List of floats
:return: Mean of values.
```

(continues on next page)

(continued from previous page)

```
.. deprecated:: 2.5.0
   Since Python 3.4, you can use the standard function
   :func:`statistics.mean`.
```

1.4.4 More elaborate example

The Deprecated library offers you 3 decorators:

- *deprecated()*: insert a `deprecated` directive in docstring, and emit a warning on each call.
- *versionadded()*: insert a `versionadded` directive in docstring, don't emit warning.
- *versionchanged()*: insert a `versionchanged` directive in docstring, don't emit warning.

The decorators can be combined to reflect the life cycle of a function:

- When it is added in your API, with the `@versionadded` decorator,
- When it has an important change, with the `@versionchanged` decorator,
- When it is deprecated, with the `@deprecated` decorator.

The example bellow illustrate this life cycle:

```
# coding: utf-8
from deprecated.sphinx import deprecated
from deprecated.sphinx import versionadded
from deprecated.sphinx import versionchanged


@deprecated(
    reason="""
    This is deprecated, really. So you need to use another function.
    But I don\'t know which one.

       - The first,
       - The second.

    Just guess!
    """,
    version='0.3.0',
)
@versionchanged(
    reason='Well, I add a new feature in this function. '
           'It is very useful as you can see in the example below, so try it. '
           'This is a very very very very very very long sentence.',
    version='0.2.0',
)
@versionadded(reason='Here is my new function.', version='0.1.0')
def successor(n):
    """
    Calculate the successor of a number.

    :param n: a number
    :return: number + 1
    """
    return n + 1
```

To see the result, you can use the builtin function `help()` to display a formatted help message of the *successor()* function. It is something like this:

```
Help on function successor in module __main__:

successor(n)
    Calculate the successor of a number.

    :param n: a number
    :return: number + 1


    .. versionadded:: 0.1.0
       Here is my new function.


    .. versionchanged:: 0.2.0
       Well, I add a new feature in this function. It is very useful as
       you can see in the example below, so try it. This is a very very
       very very very long sentence.


    .. deprecated:: 0.3.0
       This is deprecated, really. So you need to use another function.
       But I don't know which one.

          - The first,
          - The second.

       Just guess!
```

Note: Decorators must be writen in reverse order: recent first, older last.

1.4.5 Building the documentation

The easiest way to build your API documentation is to use the autodoc plugin. The directives like `automodule`, `autoclass`, `autofunction` scan your source code and generate the documentation from your docstrings.

Usually, the first thing that we need to do is indicate where the Python package that contains your source code is in relation to the `conf.py` file.

But here, that will not work! The reason is that your modules must be imported during build: the Deprecated decorators must be interpreted.

So, to build the API documentation of your project with Sphinx you need to setup a virtualenv, and install Sphinx, external themes and/or plugins and also your project. Nowadays, this is the right way to do it.

For instance, you can configure a documentation building task in your `tox.ini` file, for instance:

```
[testenv:docs]
basepython = python
deps =
    sphinx
commands =
    sphinx-build -b html -d {envtmpdir}/doctrees docs/source/ {envtmpdir}/html
```

Hint: You can see a sample implementation of Sphinx directives in the demo project Deprecated-Demo.Sphinx.

1.5 White Paper

This white paper shows some examples of how function deprecation is implemented in the Python Standard Library and Famous Open Source libraries.

You will see which kind of deprecation you can find in such libraries, and how it is documented in the user manuel.

1.5.1 The Python Standard Library

Library Python

GitHub python/cpython.

Version v3.8.dev

An example of function deprecation can be found in the `urllib` module (`Lib/urllib/parse.py`):

```
def to_bytes(url):
    warnings.warn("urllib.parse.to_bytes() is deprecated as of 3.8",
                  DeprecationWarning, stacklevel=2)
    return _to_bytes(url)
```

In the Python library, a warning is emitted in the function body using the function `warnings.warn()`. This implementation is straightforward, it uses the category `DeprecationWarning` for warning filtering.

Another example is the deprecation of the *collections* ABC, which are now moved in the `collections.abc` module. This example is available in the `collections` module (`Lib/collections/__init__.py`):

```
def __getattr__(name):
    if name in _collections_abc.__all__:
        obj = getattr(_collections_abc, name)
        import warnings
        warnings.warn("Using or importing the ABCs from 'collections' instead "
                      "of from 'collections.abc' is deprecated, "
                      "and in 3.8 it will stop working",
                      DeprecationWarning, stacklevel=2)
        globals()[name] = obj
        return obj
    raise AttributeError(f'module {__name__!r} has no attribute {name!r}')
```

The warning is only emitted when an ABC is accessed from the `collections` instead of `collections.abc` module.

We can also see an example of keyword argument deprecation in the `UserDict` class:

```
def __init__(*args, **kwargs):
    if not args:
        raise TypeError("descriptor '__init__' of 'UserDict' object "
                        "needs an argument")
    self, *args = args
    if len(args) > 1:
        raise TypeError('expected at most 1 arguments, got %d' % len(args))
```

(continues on next page)

(continued from previous page)

```
    if args:
        dict = args[0]
    elif 'dict' in kwargs:
        dict = kwargs.pop('dict')
        import warnings
        warnings.warn("Passing 'dict' as keyword argument is deprecated",
                      DeprecationWarning, stacklevel=2)
    else:
        dict = None
    self.data = {}
    if dict is not None:
        self.update(dict)
    if len(kwargs):
        self.update(kwargs)
```

Again, this implementation is straightforward: if the *dict* keyword argument is used, a warning is emitted.

Python make also use of the category `PendingDeprecationWarning` for instance in the `asyncio.tasks` module (`Lib/asyncio/tasks.py`):

```
@classmethod
def current_task(cls, loop=None):
    warnings.warn("Task.current_task() is deprecated, "
                  "use asyncio.current_task() instead",
                  PendingDeprecationWarning,
                  stacklevel=2)
    if loop is None:
        loop = events.get_event_loop()
    return current_task(loop)
```

The category `FutureWarning` is also used to emit a warning when the functions is broken and will be fixed in a "future" release. We can see for instance the method `find()` of the class `ElementTree` (`Lib/xml/etree/ElementTree.py`):

```
def find(self, path, namespaces=None):
    if path[:1] == "/":
        path = "." + path
        warnings.warn(
            "This search is broken in 1.3 and earlier, and will be "
            "fixed in a future version.  If you rely on the current "
            "behaviour, change it to %r" % path,
            FutureWarning, stacklevel=2
            )
    return self._root.find(path, namespaces)
```

As a conclusion:

- Python library uses `warnings.warn()` to emit a deprecation warning in the body of functions.
- 3 categories are used: `DeprecationWarning`, `PendingDeprecationWarning` and `FutureWarning`.
- The docstring doesn't show anything about deprecation.
- The documentation warns about some, but not all, deprecated usages.

1.5.2 The Flask Library

Library Flask

GitHub pallets/flask.

Version v1.1.dev

In the source code of Flask, we find only few deprecations: in the `app` (`flask/app.py`) and in the `helpers` (`flask/helpers.py`) modules.

In the Flask Library, like in the *Python Standard Library*, deprecation warnings are emitted during function calls. The implementation make use of the category `DeprecationWarning`.

Unlike the *Python Standard Library*, the docstring documents explicitly the deprecation. Flask uses Sphinx's deprecated directive:

The bellow example shows the deprecation of the `open_session()` method:

```
def open_session(self, request):
    """Creates or opens a new session.  Default implementation stores all
    session data in a signed cookie.  This requires that the
    :attr:`secret_key` is set.  Instead of overriding this method
    we recommend replacing the :class:`session_interface`.

    .. deprecated: 1.0
        Will be removed in 1.1. Use ``session_interface.open_session``
        instead.

    :param request: an instance of :attr:`request_class`.
    """

    warnings.warn(DeprecationWarning(
        '"open_session" is deprecated and will be removed in 1.1. Use'
        ' "session_interface.open_session" instead.'
    ))
    return self.session_interface.open_session(self, request)
```

Hint: When the function `warnings.warn()` is called with a `DeprecationWarning` instance, the instance class is used like a warning category.

The documentation also mention a `flask.exthook.ExtDeprecationWarning` (which is not found in Flask's source code):

```
Extension imports
`````````````````

Extension imports of the form ``flask.ext.foo`` are deprecated, you should use
``flask_foo``.

The old form still works, but Flask will issue a
``flask.exthook.ExtDeprecationWarning`` for each extension you import the old
way. We also provide a migration utility called `flask-ext-migrate
<https://github.com/pallets/flask-ext-migrate>`_ that is supposed to
automatically rewrite your imports for this.
```

As a conclusion:

- Flask library uses `warnings.warn()` to emit a deprecation warning in the body of functions.
```

- Only one category is used: `DeprecationWarning`.
- The docstring use Sphinx's deprecated directive.
- The API documentation contains the deprecated usages.

1.5.3 The Django Library

Library Django

GitHub django/django.

Version v3.0.dev

The Django Library defines several categories for deprecation in the module `django.utils.deprecation`:

- The category `RemovedInDjango31Warning` which inherits from `DeprecationWarning`.
- The category `RemovedInDjango40Warning` which inherits from `PendingDeprecationWarning`.
- The category `RemovedInNextVersionWarning` which is an alias of `RemovedInDjango40Warning`.

The Django Library don't use `DeprecationWarning` or `PendingDeprecationWarning` directly, but always use one of this 2 classes. The category `RemovedInNextVersionWarning` is only used in unit tests.

There are a lot of class deprecation examples. The deprecation warning is emitted during the call of the `__init__` method. For instance in the class `FloatRangeField` (`django/contrib/staticfiles/storage.py`):

```
class FloatRangeField(DecimalRangeField):
    base_field = forms.FloatField

    def __init__(self, **kwargs):
        warnings.warn(
            'FloatRangeField is deprecated in favor of DecimalRangeField.',
            RemovedInDjango31Warning, stacklevel=2,
        )
        super().__init__(**kwargs)
```

The implementation in the Django Library is similar to the one done in the *Python Standard Library*: deprecation warnings are emitted during function calls. The implementation use the category `RemovedInDjango31Warning`.

In the Django Library, we also find an example of property deprecation: The property `FILE_CHARSET()` of the class `django.conf.LazySettings`. The implementation of this property is:

```
@property
def FILE_CHARSET(self):
    stack = traceback.extract_stack()
    # Show a warning if the setting is used outside of Django.
    # Stack index: -1 this line, -2 the caller.
    filename, _line_number, _function_name, _text = stack[-2]
    if not filename.startswith(os.path.dirname(django.__file__)):
        warnings.warn(
            FILE_CHARSET_DEPRECATED_MSG,
            RemovedInDjango31Warning,
            stacklevel=2,
        )
    return self.__getattr__('FILE_CHARSET')
```

We also find function deprecations, mainly with the category `RemovedInDjango40Warning`. For instance, the function `smart_text()` emits a deprecation warning as follow:

```
def smart_text(s, encoding='utf-8', strings_only=False, errors='strict'):
    warnings.warn(
        'smart_text() is deprecated in favor of smart_str().',
        RemovedInDjango40Warning, stacklevel=2,
    )
    return smart_str(s, encoding, strings_only, errors)
```

The Django Library also define a decorator `warn_about_renamed_method` which is used internally in the metaclass `RenameMethodsBase`. This metaclass is only used in unit tests to check renamed methods.

As a conclusion:

- The Django library uses `warnings.warn()` to emit a deprecation warning in the body of functions.
- It uses two categories which inherits the standard categories `DeprecationWarning` and `PendingDeprecationWarning`.
- The source code of the Django Library doesn't contains much docstring. The deprecation never appears in the docstring anyway.
- The release notes contain information about deprecated features.

1.5.4 The lxml Library

Library lxml

GitHub lxml/lxml.

Version v4.3.2.dev

The lxml Library is developed in Cython, not Python. But, it is a similar language. This library mainly use comments or docstring to mark function as deprecated.

For instance, in the class `lxml.xpath._XPathEvaluatorBase`(:file:`src/lxml/xpath.pxi`), the `evaluate` method is deprecated as follow:

```
def evaluate(self, _eval_arg, **_variables):
    u"""evaluate(self, _eval_arg, **_variables)

    Evaluate an XPath expression.

    Instead of calling this method, you can also call the evaluator object
    itself.

    Variables may be provided as keyword arguments.  Note that namespaces
    are currently not supported for variables.

    :deprecated: call the object, not its method.
    """
    return self(_eval_arg, **_variables)
```

There is only one example of usage of the function `warnings.warn()`: in the `_ElementTree` class (`src/lxml/etree.pyx`):

```
if docstring is not None and doctype is None:
    import warnings
    warnings.warn(
        "The 'docstring' option is deprecated. Use 'doctype' instead.",
```

(continues on next page)

(continued from previous page)

```
        DeprecationWarning)
    doctype = docstring
```

As a conclusion:

- Except in one example, the lxml library doesn't use `warnings.warn()` to emit a deprecation warnings.
- The deprecations are described in the function docstrings.
- The release notes contain information about deprecated features.

1.5.5 The openpyxl Library

Library openpyxl

Bitbucket openpyxl/openpyxl.

Version v2.6.1.dev

openpyxl is a Python library to read/write Excel 2010 xlsx/xlsm/xltx/xltm files. Tu warn about deprecation, this library uses a home-made `@deprecated` decorator.

The implementation of this decorator is an adapted copy of the first version of Tantale's `@deprecated` decorator. It has the enhancement to update the docstring of the decorated function. So, this is similar to the function *deprecated.sphinx.deprecated()*.

```
string_types = (type(b''), type(u''))
def deprecated(reason):

    if isinstance(reason, string_types):

        def decorator(func1):

            if inspect.isclass(func1):
                fmt1 = "Call to deprecated class {name} ({reason})."
            else:
                fmt1 = "Call to deprecated function {name} ({reason})."

            @wraps(func1)
            def new_func1(*args, **kwargs):
                #warnings.simplefilter('default', DeprecationWarning)
                warnings.warn(
                    fmt1.format(name=func1.__name__, reason=reason),
                    category=DeprecationWarning,
                    stacklevel=2
                )
                return func1(*args, **kwargs)

            # Enhance docstring with a deprecation note
            deprecationNote = "\n\n.. note::\n    Deprecated: " + reason
            if new_func1.__doc__:
                new_func1.__doc__ += deprecationNote
            else:
                new_func1.__doc__ = deprecationNote
            return new_func1

        return decorator
```

(continues on next page)

(continued from previous page)

```
    elif inspect.isclass(reason) or inspect.isfunction(reason):
        raise TypeError("Reason for deprecation must be supplied")

    else:
        raise TypeError(repr(type(reason)))
```

As a conclusion:

- The openpyxl library uses a decorator to deprecate functions.
- It uses the category `DeprecationWarning`.
- The decorator update the docstring and add a `.. note::` directive, which is visible in the documentation.

CHAPTER 2

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

2.1 API

This part of the documentation covers all the interfaces of the Deprecated Library.

2.1.1 Deprecated Library

Python `@deprecated` decorator to deprecate old python classes, functions or methods.

2.1.2 Classic deprecation warning

Classic `@deprecated` decorator to deprecate old python classes, functions or methods.

class deprecated.classic.**ClassicAdapter**(*reason=''*, *version=''*, *action=None*, *category=<class 'DeprecationWarning'>*, *extra_stacklevel=0*)

Classic adapter – *for advanced usage only*

This adapter is used to get the deprecation message according to the wrapped object type: class, function, standard method, static method, or class method.

This is the base class of the *SphinxAdapter* class which is used to update the wrapped object docstring.

You can also inherit this class to change the deprecation message.

In the following example, we change the message into "The … is deprecated.":

```
import inspect

from deprecated.classic import ClassicAdapter
```

(continues on next page)

(continued from previous page)

```
from deprecated.classic import deprecated


class MyClassicAdapter(ClassicAdapter):
    def get_deprecated_msg(self, wrapped, instance):
        if instance is None:
            if inspect.isclass(wrapped):
                fmt = "The class {name} is deprecated."
            else:
                fmt = "The function {name} is deprecated."
        else:
            if inspect.isclass(instance):
                fmt = "The class method {name} is deprecated."
            else:
                fmt = "The method {name} is deprecated."
        if self.reason:
            fmt += " ({reason})"
        if self.version:
            fmt += " -- Deprecated since version {version}."
        return fmt.format(name=wrapped.__name__,
                          reason=self.reason or "",
                          version=self.version or "")
```

Then, you can use your `MyClassicAdapter` class like this in your source code:

```
@deprecated(reason="use another function", adapter_cls=MyClassicAdapter)
def some_old_function(x, y):
    return x + y
```

get_deprecated_msg(*wrapped*, *instance*)

Get the deprecation warning message for the user.

Parameters

- **wrapped** – Wrapped class or function.
- **instance** – The object to which the wrapped function was bound when it was called.

Returns The warning message.

deprecated.classic.**deprecated**(**args*, ***kwargs*)

This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

Classic usage:

To use this, decorate your deprecated function with **@deprecated** decorator:

```
from deprecated import deprecated


@deprecated
def some_old_function(x, y):
    return x + y
```

You can also decorate a class or a method:

```
from deprecated import deprecated
```

(continues on next page)

(continued from previous page)

```
class SomeClass(object):
    @deprecated
    def some_old_method(self, x, y):
        return x + y


@deprecated
class SomeOldClass(object):
    pass
```

You can give a *reason* message to help the developer to choose another function/class, and a *version* number to specify the starting version number of the deprecation.

```
from deprecated import deprecated


@deprecated(reason="use another function", version='1.2.0')
def some_old_function(x, y):
    return x + y
```

The *category* keyword argument allow you to specify the deprecation warning class of your choice. By default, `DeprecationWarning` is used, but you can choose `FutureWarning`, `PendingDeprecationWarning` or a custom subclass.

```
from deprecated import deprecated


@deprecated(category=PendingDeprecationWarning)
def some_old_function(x, y):
    return x + y
```

The *action* keyword argument allow you to locally change the warning filtering. *action* can be one of "error", "ignore", "always", "default", "module", or "once". If `None`, empty or missing, the global filtering mechanism is used. See: The Warnings Filter in the Python documentation.

```
from deprecated import deprecated


@deprecated(action="error")
def some_old_function(x, y):
    return x + y
```

The *extra_stacklevel* keyword argument allows you to specify additional stack levels to consider instrumentation rather than user code. With the default value of 0, the warning refers to where the class was instantiated or the function was called.

2.1.3 Sphinx directive integration

We usually need to document the life-cycle of functions and classes: when they are created, modified or deprecated.

To do that, Sphinx has a set of Paragraph-level markups:

- `versionadded`: to document the version of the project which added the described feature to the library,
- `versionchanged`: to document changes of a feature,

- `deprecated`: to document a deprecated feature.

The purpose of this module is to defined decorators which adds this Sphinx directives to the docstring of your function and classes.

Of course, the `@deprecated` decorator will emit a deprecation warning when the function/method is called or the class is constructed.

class deprecated.sphinx.**SphinxAdapter**(*directive*, *reason=''*, *version=''*, *action=None*, *category=<class 'DeprecationWarning'>*, *extra_stacklevel=0*, *line_length=70*)

Sphinx adapter – *for advanced usage only*

This adapter override the *ClassicAdapter* in order to add the Sphinx directives to the end of the function/class docstring. Such a directive is a Paragraph-level markup

- The directive can be one of "versionadded", "versionchanged" or "deprecated".
- The version number is added if provided.
- The reason message is obviously added in the directive block if not empty.

get_deprecated_msg(*wrapped*, *instance*)

Get the deprecation warning message (without Sphinx cross-referencing syntax) for the user.

Parameters

- **wrapped** – Wrapped class or function.
- **instance** – The object to which the wrapped function was bound when it was called.

Returns The warning message.

New in version 1.2.12: Strip Sphinx cross-referencing syntax from warning message.

deprecated.sphinx.**deprecated**(*reason=''*, *version=''*, *line_length=70*, ***kwargs*)

This decorator can be used to insert a "deprecated" directive in your function/class docstring in order to document the version of the project which deprecates this functionality in your library.

Parameters

- **reason** (*str*) – Reason message which documents the deprecation in your library (can be omitted).
- **version** (*str*) – Version of your project which deprecates this feature. If you follow the Semantic Versioning, the version number has the format "MAJOR.MINOR.PATCH".
- **line_length** (*int*) – Max line length of the directive text. If non nul, a long text is wrapped in several lines.

Keyword arguments can be:

- "action": A warning filter used to activate or not the deprecation warning. Can be one of "error", "ignore", "always", "default", "module", or "once". If `None`, empty or missing, the global filtering mechanism is used.
- "category": The warning category to use for the deprecation warning. By default, the category class is `DeprecationWarning`, you can inherit this class to define your own deprecation warning category.
- "extra_stacklevel": Number of additional stack levels to consider instrumentation rather than user code. With the default value of 0, the warning refers to where the class was instantiated or the function was called.

Returns a decorator used to deprecate a function.

Changed in version 1.2.13: Change the signature of the decorator to reflect the valid use cases.

Changed in version 1.2.15: Add the *extra_stacklevel* parameter.

deprecated.sphinx.**versionadded**(*reason=''*, *version=''*, *line_length=70*)

This decorator can be used to insert a "versionadded" directive in your function/class docstring in order to document the version of the project which adds this new functionality in your library.

Parameters

- **reason** (*str*) – Reason message which documents the addition in your library (can be omitted).
- **version** (*str*) – Version of your project which adds this feature. If you follow the Semantic Versioning, the version number has the format "MAJOR.MINOR.PATCH", and, in the case of a new functionality, the "PATCH" component should be "0".
- **line_length** (*int*) – Max line length of the directive text. If non nul, a long text is wrapped in several lines.

Returns the decorated function.

deprecated.sphinx.**versionchanged**(*reason=''*, *version=''*, *line_length=70*)

This decorator can be used to insert a "versionchanged" directive in your function/class docstring in order to document the version of the project which modifies this functionality in your library.

Parameters

- **reason** (*str*) – Reason message which documents the modification in your library (can be omitted).
- **version** (*str*) – Version of your project which modifies this feature. If you follow the Semantic Versioning, the version number has the format "MAJOR.MINOR.PATCH".
- **line_length** (*int*) – Max line length of the directive text. If non nul, a long text is wrapped in several lines.

Returns the decorated function.

CHAPTER 3

Additional Notes

Legal information and changelog are here for the interested.

3.1 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

Note: The library **"Python-Deprecated"** was renamed **"Deprecated"**, simply! This project is more consistent because now, the name of the library is the same as the name of the Python package.

- In your `setup.py`, you can replace the "Python-Deprecated" dependency with "Deprecated".
- In your source code, nothing has changed, you will always use `import deprecated`, as before.
- I decided to keep the same version number because there is really no change in the source code (only in comment or documentation).

3.1.1 v1.2.15 (unreleased)

Bug fix release

Fix

- Resolve Python 2.7 support issue introduced in v1.2.14 in `sphinx.py`.
- Fix #69: Add `extra_stacklevel` argument for interoperating with other wrapper functions (refer to #68 for a concrete use case).

Other

- Fix #66: discontinue TravisCI and AppVeyor due to end of free support.

3.1.2 v1.2.14 (2023-05-27)

Bug fix release

Fix

- Fix #60: return a correctly dedented docstring when long docstring are using the D212 or D213 format.

Other

- Add support for Python 3.11.
- Drop support for Python older than 3.7 in build systems like pytest and tox, while ensuring the library remains production-compatible.
- Update GitHub workflow to run in recent Python versions.

3.1.3 v1.2.13 (2021-09-05)

Bug fix release

Fix

- Fix #45: Change the signature of the *deprecated()* decorator to reflect the valid use cases.
- Fix #48: Fix `versionadded` and `versionchanged` decorators: do not return a decorator factory, but a Wrapt adapter.

Other

- Fix configuration for AppVeyor: simplify the test scripts and set the version format to match the current version.
- Change configuration for Tox:
  - change the requirements for `pip` to "pip >= 9.0.3, < 21" (Python 2.7, 3.4 and 3.5).
  - install `typing` when building on Python 3.4 (required by Pytest->Attrs).
  - run unit tests on Wrapt 1.13 (release candidate).
- Migrating project to travis-ci.com.

3.1.4 v1.2.12 (2021-03-13)

Bug fix release

Fix

- Avoid "Explicit markup ends without a blank line" when the decorated function has no docstring.
- Fix #40: 'version' argument is required in Sphinx directives.
- Fix #41: *deprecated.sphinx*: strip Sphinx cross-referencing syntax from warning message.

Other

- Change in Tox and Travis CI configurations: enable unit testing on Python 3.10.

3.1.5 v1.2.11 (2021-01-17)

Bug fix release

Fix

- Fix packit configuration: use `upstream_tag_template: v{version}`.
- Fix #33: Change the class *SphinxAdapter*: add the `line_length` keyword argument to the constructor to specify the max line length of the directive text. Sphinx decorators also accept the `line_length` argument.
- Fix #34: `versionadded` and `versionchanged` decorators don't emit `DeprecationWarning` anymore on decorated classes.

Other

- Change the Tox configuration to run tests on Python 2.7, Python 3.4 and above (and PyPy 2.7 & 3.6).
- Update the classifiers in `setup.py`.
- Replace `bumpversion` by bump2version in `setup.py` and documentation.
- Update configuration for Black and iSort.
- Fix the development requirement versions in `setup.py` for Python 2.7 EOL.

3.1.6 v1.2.10 (2020-05-13)

Bug fix release

Fix

- Fix #25: `@deprecated` respects global warning filters with actions other than "ignore" and "always" on Python 3.

Other

- Change the configuration for TravisCI to build on pypy and pypy3.
- Change the configuration for TravisCI and AppVeyor: drop configuration for Python **3.4** and add **3.8**.

3.1.7 v1.2.9 (2020-04-10)

Bug fix release

Fix

- Fix #20: Set the `warnings.warn()` stacklevel to 2 if the Python implementation is PyPy.
- Fix packit configuration: use `dist-git-branch: fedora-all`.

Other

- Change the Tox configuration to run tests on PyPy v2.7 and 3.6.

3.1.8 v1.2.8 (2020-04-05)

Bug fix release

Fix

- Fix #15: The `@deprecated` decorator doesn't set a warning filter if the *action* keyword argument is not provided or `None`. In consequences, the warning messages are only emitted if the global filter allow it. For more information, see The Warning Filter in the Python documentation.
- Fix #13: Warning displays the correct filename and line number when decorating a class if wrapt does not have the compiled c extension.

Documentation

- The *API* documentation and the *Tutorial* is improved to explain how to use custom warning categories and local filtering (warning filtering at function call).
- Fix #17: Customize the sidebar to add links to the documentation to the source in GitHub and to the Bug tracker. Add a logo in the sidebar and change the logo in the main page to see the library version.
- Add a detailed documentation about *The "Sphinx" decorators*.

Other

- Change the Tox configuration to test the library with Wrapt 1.12.x.

3.1.9 v1.2.7 (2019-11-11)

Bug fix release

Fix

- Fix #13: Warning displays the correct filename and line number when decorating a function if wrapt does not have the compiled c extension.

Other

- Support packit for Pull Request tests and sync to Fedora (thanks to Petr Hráček). Supported since v1.2.6.
- Add Black configuration file.

3.1.10 v1.2.6 (2019-07-06)

Bug fix release

Fix

- Fix #9: Change the project's configuration: reinforce the constraint to the Wrapt requirement.

Other

- Upgrade project configuration (`setup.py`) to add the *project_urls* property: Documentation, Source and Bug Tracker URLs.
- Change the Tox configuration to test the library against different Wrapt versions.
- Fix an issue with the AppVeyor build: upgrade setuptools version in `appveyor.yml`, change the Tox configuration: set `py27,py34,py35:  pip >= 9.0.3, < 19.2`.

3.1.11 v1.2.5 (2019-02-28)

Bug fix release

Fix

- Fix #6: Use `inspect.isroutine()` to check if the wrapped object is a user-defined or built-in function or method.

Other

- Upgrade Tox configuration to add support for Python 3.7. Also, fix PyTest version for Python 2.7 and 3.4 (limited support). Remove dependency 'requests[security]': useless to build documentation.
- Upgrade project configuration (`setup.py`) to add support for Python 3.7.

3.1.12 v1.2.4 (2018-11-03)

Bug fix release

Fix

- Fix #4: Correct the class *`ClassicAdapter`*: Don't pass arguments to `object.__new__()` (other than *cls*).

Other

- Add missing docstring to the classes *ClassicAdapter* and *SphinxAdapter*.
- Change the configuration for TravisCI and AppVeyor: drop configuration for Python **2.6** and **3.3**. add configuration for Python **3.7** (if available).

Note: Deprecated is no more tested with Python **2.6** and **3.3**. Those Python versions are EOL for some time now and incur incompatibilities with Continuous Integration tools like TravisCI and AppVeyor. However, this library should still work perfectly...

3.1.13 v1.2.3 (2018-09-12)

Bug fix release

Fix

- Fix #3: `deprecated.sphinx` decorators don't update the docstring.

3.1.14 v1.2.2 (2018-09-04)

Bug fix release

Fix

- Fix #2: a deprecated class is a class (not a function). Any subclass of a deprecated class is also deprecated.
- Minor fix: add missing documentation in *deprecated.sphinx* module.

3.1.15 v1.2.1 (2018-08-27)

Bug fix release

Fix

- Add a `MANIFEST.in` file to package additional files like "LICENSE.rst" in the source distribution.

3.1.16 v1.2.0 (2018-04-02)

Minor release

Added

- Add decorators for Sphinx directive integration: `versionadded`, `versionchanged`, `deprecated`. That way, the developer can document the changes.

Changed

- Add the `version` parameter to the `@deprecated` decorator: used to specify the starting version number of the deprecation.
- Add a way to choose a `DeprecationWarning` subclass.

Removed

- Deprecated no longer supports Python **2.6** and **3.3**. Those Python versions are EOL for some time now and incur maintenance and compatibility costs on the Deprecated core team, and following up with the rest of the community we decided that they will no longer be supported starting on this version. Users which still require those versions should pin Deprecated to `< 1.2`.

3.1.17 v1.1.5 (2019-02-28)

Bug fix release

Fix

- Fix #6: Use `inspect.isroutine()` to check if the wrapped object is a user-defined or built-in function or method.

Other

- Upgrade Tox configuration to add support for Python 3.7. Also, fix PyTest version for Python 2.7 and 3.4 (limited support). Remove dependency 'requests[security]': useless to build documentation.
- Upgrade project configuration (`setup.py`) to add support for Python 3.7.

3.1.18 v1.1.4 (2018-11-03)

Bug fix release

Fix

- Fix #4: Correct the function `deprecated()`: Don't pass arguments to `object.__new__()` (other than *cls*).

Other

- Change the configuration for TravisCI and AppVeyor: drop configuration for Python **2.6** and **3.3**. add configuration for Python **3.7**.

Note: Deprecated is no more tested with Python **2.6** and **3.3**. Those Python versions are EOL for some time now and incur incompatibilities with Continuous Integration tools like TravisCI and AppVeyor. However, this library should still work perfectly...

3.1.19 v1.1.3 (2018-09-03)

Bug fix release

Fix

- Fix #2: a deprecated class is a class (not a function). Any subclass of a deprecated class is also deprecated.

3.1.20 v1.1.2 (2018-08-27)

Bug fix release

Fix

- Add a `MANIFEST.in` file to package additional files like "LICENSE.rst" in the source distribution.

3.1.21 v1.1.1 (2018-04-02)

Bug fix release

Fix

- Minor correction in `CONTRIBUTING.rst` for Sphinx builds: add the `-d` option to put apart the `doctrees` from the generated documentation and avoid warnings with epub generator.
- Fix in documentation configuration: remove hyphens in `epub_identifier` (ISBN number has no hyphens).
- Fix in Tox configuration: set the versions interval of each dependency.

Other

- Change in documentation: improve sentence phrasing in the Tutorial.
- Restore the epub title to "Python Deprecated Library v1.1 Documentation" (required for Lulu.com).

3.1.22 v1.1.0 (2017-11-06)

Minor release

Added

- Change in `deprecated.deprecated()` decorator: you can give a "reason" message to help the developer choose another class, function or method.
- Add support for Universal Wheel (Python versions 2.6, 2.7, 3.3, 3.4, 3.5, 3.6 and PyPy).
- Add missing `__doc__` and `__version__` attributes to *deprecated* module.
- Add an extensive documentation of Deprecated Library.

Other

- Improve Travis configuration file (compatibility from Python 2.6 to 3.7-dev, and PyPy).
- Add AppVeyor configuration file.
- Add Tox configuration file.
- Add BumpVersion configuration file.
- Improve project settings: add a long description for the project. Set the **license** and the **development status** in the classifiers property.
- Add the `CONTRIBUTING.rst` file: "How to contribute to Deprecated Library".

3.1.23 v1.0.0 (2016-08-30)

Major release

Added

- **deprecated**: Created **@deprecated** decorator

3.2 License

The MIT License (MIT)

Copyright (c) 2017 Laurent LAPORTE

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.

3.3 How to contribute to Deprecated Library

Thank you for considering contributing to Deprecated!

3.3.1 Support questions

Please, don't use the issue tracker for this. Use one of the following resources for questions about your own code:

- Ask on Stack Overflow. Search with Google first using: `site:stackoverflow.com deprecated decorator {search term, exception message, etc.}`

3.3.2 Reporting issues

- Describe what you expected to happen.
- If possible, include a minimal, complete, and verifiable example to help us identify the issue. This also helps check that the issue is not with your own code.
- Describe what actually happened. Include the full traceback if there was an exception.
- List your Python, Deprecated versions. If possible, check if this issue is already fixed in the repository.

3.3.3 Submitting patches

- Include tests if your patch is supposed to solve a bug, and explain clearly under which circumstances the bug happens. Make sure the test fails without your patch.
- Try to follow PEP8, but you may ignore the line length limit if following it would make the code uglier.

First time setup

- Download and install the latest version of git.
- Configure git with your username and email:

```
git config --global user.name 'your name'
git config --global user.email 'your email'
```

- Make sure you have a GitHub account.
- Fork Deprecated to your GitHub account by clicking the Fork button.
- Clone your GitHub fork locally:

```
git clone https://github.com/{username}/deprecated.git
cd deprecated
```

- Add the main repository as a remote to update later:

```
git remote add tantale https://github.com/tantale/deprecated.git
git fetch tantale
```

- Create a virtualenv:

```
python3 -m venv env
. env/bin/activate
# or "env\Scripts\activate" on Windows
```

- Install Deprecated in editable mode with development dependencies:

```
pip install -e ".[dev]"
```

Start coding

- Create a branch to identify the issue you would like to work on (e.g. `2287-dry-test-suite`)
- Using your favorite editor, make your changes, committing as you go.
- Try to follow PEP8, but you may ignore the line length limit if following it would make the code uglier.
- Include tests that cover any code changes you make. Make sure the test fails without your patch. *Running the tests*.
- Push your commits to GitHub and create a pull request.
- Celebrate

Running the tests

Run the basic test suite with:

```
pytest tests/
```

This only runs the tests for the current environment. Whether this is relevant depends on which part of Deprecated you're working on. Travis-CI will run the full suite when you submit your pull request.

The full test suite takes a long time to run because it tests multiple combinations of Python and dependencies. You need to have Python 2.7, 3.4, 3.5, 3.6, PyPy 2.7 and 3.6 installed to run all of the environments (notice that Python **2.6** and **3.3** are no more supported). Then run:

```
tox
```

Running test coverage

Generating a report of lines that do not have test coverage can indicate where to start contributing. Run `pytest` using `coverage` and generate a report on the terminal and as an interactive HTML document:

```
pytest --cov-report term-missing --cov-report html --cov=deprecated tests/
# then open htmlcov/index.html
```

Read more about coverage.

Running the full test suite with `tox` will combine the coverage reports from all runs.

`make` targets

Deprecated provides a `Makefile` with various shortcuts. They will ensure that all dependencies are installed.

- `make test` runs the basic test suite with `pytest`
- `make cov` runs the basic test suite with `coverage`
- `make test-all` runs the full test suite with `tox`

Generating the documentation

The documentation is automatically generated with ReadTheDocs for each git push on master. You can also generate it manually using Sphinx.

To generate the HTML documentation, run:

```
sphinx-build -b html -d dist/docs/doctrees docs/source/ dist/docs/html/
```

To generate the epub v2 documentation, run:

```
sphinx-build -b epub -d dist/docs/doctrees docs/source/ dist/docs/epub/
```

Python Module Index

d

Index