Auto deprecator Documentation

Release 2020.5.0

Gavin Chan

Aug 27, 2020

Contents:

CHAPTER 1

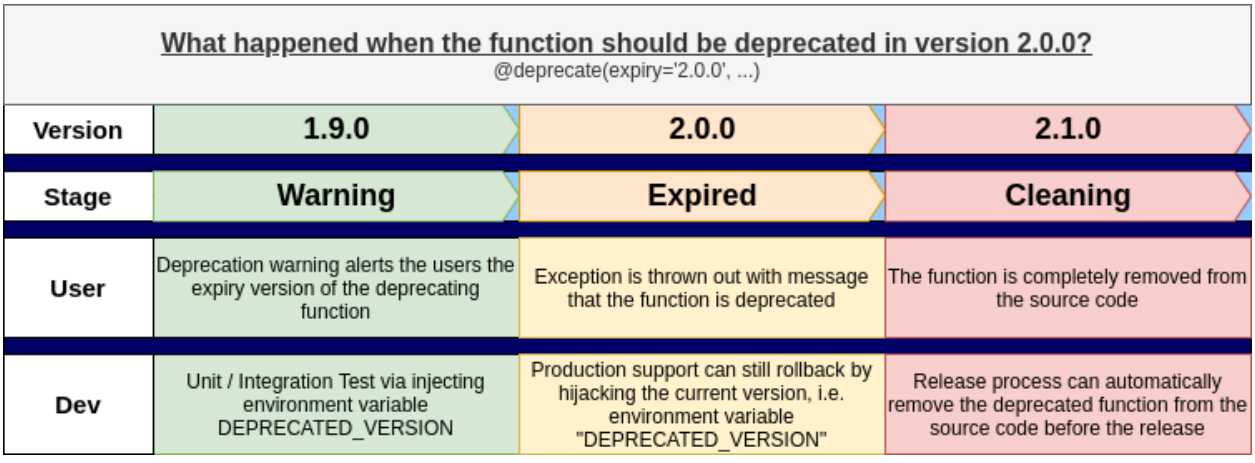
Auto deprecator

Deprecation toolkit in Python

- Free software: MIT license
- Documentation: https://auto-deprecator.readthedocs.io.

1.1 How does it work?

The library provides the full cycle to deprecate a function in the following ways



For example, a function called `old_hello_world` should be deprecated in the version 2.0.0, while the current version of the library is 1.0.0.

Add a decorator `deprecate` above the function like the below can manage the mentioned deprecation cycle.

```
from auto_deprecator import deprecate


@deprecate(expiry='2.0.0', current='1.9.0')
def old_hello_world():
    return print("Hello world!")


def hello_world():
    return print("Hello world again!")
```

You can also suggest the replacing function / method. For details, please refer to the section *Provide hints to users*.

1.1.1 Warning Stage

Alert the users the deprecation time

When the user calls the methods or initializes the objects which will be deprecated in the next version or on an expected date, the user should receive the warning of the future deprecation but get the return in success. The default warning handler is to throw a `DeprecationWarning` and the handle method can be customized in the section *Customize the deprecation handling*

```
>>> old_hello_world()
Hello world!
DeprecationWarning: The function "old_hello_world" will be deprecated on version 2.0.0
```

Test as if deprecated

Before the component is deprecated, unit / integration testing should be run to ensure the deprecation does not break the existing flow. Pass in the environment variables in the testing to simulate that the version is deployed.

```
(bash) hello-world-app
Hello world!
DeprecationWarning: The function "old_hello_world" will be deprecated in version 2.0.0
```

```
(bash) DEPRECATED_VERSION=2.0.0 hello-world-app
Traceback (most recent call last):
 ...
RuntimeError: The function "old_hello_world" is deprecated in version 2.0.0
```

1.1.2 Expired Stage

If the current version has reached the function expiry version, calling the deprecated function will trigger the exception by default.

```
from auto_deprecator import deprecate
```

(continues on next page)

(continued from previous page)

```
__version__ = '2.0.0'


@deprecate(expiry='2.0.0', current=__version__)
def old_hello_world():
    return print("Hello world!")
```

For example, the above function is called by the downstream process `after-hello-world`. The owner of the process is not aware that the function should be deprecated and replaced by another function, and the process is crashed by the default exception. To work around the exception in the production, before a proper fix is provided, the environment variable `DEPRECATED_VERSION` can be injected in the downstream process.

```
DEPRECATED_VERSION=1.9 after-hello-world
```

1.1.3 Cleaning Stage

Automatic deprecation before release

Deprecating the functions is no longer a manual work. Every time before release, run the command `auto-deprecate` to remove the functions deprecated in the coming version.

```
$ auto-deprecate hello_world.py --version 2.0.0
```

The command removes the function `old_hello_world` from the source codes as the expiry version is 2.0.0. Also, if the source file does not require to import the `auto-deprecate` anymore (as all the functions have already been deprecated), the import line will be removed as well.

```
$ git difftool -y -x sdiff
from auto_deprecator import deprecate                       <
                                                            <
                                                            <
@deprecate(expiry='2.0.0', current='1.9.0')                 <
def old_hello_world():                                      <
    return print("Hello world!")                            <
                                                            <
                                                            <
def hello_world():                                            def hello_world():
    return print("Hello world again!")                     /     return print(
↪"Hello world again!")
```

The function with a comment line to state the expiry version is another way to inform the script `auto-deprecate` to remove the part of the code when it is expired. For example,

```
def old_hello_world():
    # auto-deprecate: expiry=2.0.0
    print('hello world')
```

For the details of the comment hints, please refer to the section *Auto deprecation hints in comments*.

1.2 Installation

The library can be easily installed with pip

```
pip install auto-deprecator
```

1.3 Alternative Installation

If the auto-deprecator is included and the functions are well deprecated (following the whole cycle mentioned above), your software does not need auto-deprecator anymore. For developers who are not comfortable to include a library not always in use as a dependency, they can just clone the source code into your project instead.

For example, your Python project contains a module called "utils" to maintain all the utility functions.

```
.
├── setup.py
└── test_py_project
    ├── cli.py
    ├── __init__.py
    ├── test_py_project.py
    └── utils
        └── __init__.py
```

With the bash command "curl",

```
curl https://raw.githubusercontent.com/auto-deprecator/auto-deprecator/develop/auto_
↪deprecator/__init__.py -o $DEST
```

the source code of auto-deprecator can be cloned into the target directory, i.e. "test_py_project/utils" in the example

```
curl https://raw.githubusercontent.com/auto-deprecator/auto-deprecator/develop/auto_
↪deprecator/__init__.py -o test_py_project/utils/auto_deprecator.py
```

1.4 Features

1.4.1 Provide hints to users

Provide the parameter "relocate", the warning / error message will inform the user about the relocated method.

```
@deprecate(expiry='2.1.0', current='2.0.0', relocate='new_compute_method')
def compute_method():
    return 'hello world'
```

```
>>> old_hello_world()
Hello world!
DeprecationWarning: The function "old_hello_world" will be deprecated on version 2.
↪0..
                    Please use method / function "new_compute_method".
```

1.4.2 Import current version from module name

Instead of importing the version (`__version__`) in the module,

```
from your_package import __version__

@deprecate(expiry='2.1.0', current=__version__)
def compute_method():
    return 'hello world'
```

specifying the module name, which includes the version attribute, can help maintain the source code in a clean manner.

```
@deprecate(expiry='2.1.0', version_module='your_package')
def compute_method():
    return 'hello world'
```

Especially if the function is removed by the action `auto-deprecate`, the unused import will not be left in the module.

1.4.3 Customize the deprecation handling

By default, the `deprecate` decorator raise `DeprecationWarning` for the future expiry and `RuntimeError` on the expiration. The behavior can be modified so as to fit in the infrastructure / production environment.

For example, the `DeprecationWarning` can be replaced by a simple print out by injecting a callable function into the parameter `warn_handler`.

```
@deprecate(expiry='2.1.0', current='2.0.0', warn_handler=print)
def compute_method():
    return 'hello world'
```

Same for injecting a callable function into the parameter `error_handler`, the behavior is replaced if the function is deprecated.

1.4.4 Auto deprecation hints in comments

The auto deprecation script handles not only the expiry parts wrapped by the decorator, but also those stated with comments. The comment line in the format `# auto-deprecate: expiry=<version>` in the scope of the function or class is treated same as the decorator hints `@deprecate(expiry="version", ...)`.

For example, the below function will be removed

```
# hello_world.py


def old_hello_world():
    # auto-deprecate: expiry=2.0.0
    print('hello world')
```

when the script is called with current version greater than 2.0.0

```
$ auto-deprecate hello_world.py --version 2.1.0
```

CHAPTER 2

Installation

2.1 Stable release

To install Auto deprecator, run this command in your terminal:

```
$ pip install auto_deprecator
```

This is the preferred method to install Auto deprecator, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for Auto deprecator can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/gavincyi/auto_deprecator
```

Or download the tarball:

```
$ curl -OJL https://github.com/gavincyi/auto_deprecator/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use Auto deprecator in a project:

```
import auto_deprecator
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/gavincyi/auto_deprecator/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.4 Write Documentation

Auto deprecator could always use more documentation, whether as part of the official Auto deprecator docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/gavincyi/auto_deprecator/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *auto_deprecator* for local development.

1. Fork the *auto_deprecator* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/auto_deprecator.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv auto_deprecator
$ cd auto_deprecator/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 auto_deprecator tests
$ python setup.py test or pytest
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.5, 3.6, 3.7 and 3.8, and for PyPy. Check https://travis-ci.org/gavincyi/auto_deprecator/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ pytest tests.test_auto_deprecator
```

4.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bump2version patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 5

History

5.1 2020.4.0 (2020-04-23)

- Support deprecation hints in comments
- Simplify the project architecture

5.2 2020.3.0 (2020-04-11)

- Support automatic deprecation in the directory
- Support customizing the deprecation handler

5.3 2020.2.0 (2020-02-11)

- Introduce parameter version_module in the deprecate decorator, to import the version dynamically
- Removed magic version import

5.4 2010.1.0, 2010.1.1, 2010.1.2 (2020-01-21)

- Support alerting the users the deprecate version
- Support testing with environment variables
- Support automatically deprecate the expiry source code

CHAPTER 6

Indices and tables

- genindex
- modindex
- search