# Fighting Documentation Drift

**Terrence Reilly**

terrencepreilly@gmail.com

## 1. The Problem

The tendency for documentation to become out-of-date with the code it describes is known as *documentation drift*. Documentation drift can result in lost time, incorrect use of APIs, and frustration.

The following example of documentation drift is taken from a convenience method in the project *darglint*. The parameter *filename* was added to the function, but the docstring was not updated. In this particular example, it is not entirely obvious how *filename* is supposed to be used simply from reading the function.

```
def get_error_report(self, verbosity, filename):
    """Return a string representation of the errors.

    Args:
        verbosity:
            The level of verbosity.  Should be an
            integer in the range [1,3].

    Returns:
        A string representation of the errors.

    """
    return str(ErrorReport(
        errors=self.errors,
        filename=filename,
        verbosity=verbosity,
    ))
```

## 2. Typical Solutions to Documentation Drift

As with all risks, the dangers of documentation drift can be mitigated using three general strategies: assessment, avoidance, and control. The most common strategy used by developers is avoidance.

On the extreme end, developers can avoid out-of-sync documentation by not writing documentation to begin with. Proponents of avoidance argue that documentation is stale the moment it is written, and that code is the only source of truth. Advocates frequently argue that code should be self-documenting, and that the need for documentation is a code smell.

While clarity is a good quality for software to have, relying solely on source code as documentation leaves out the benefits of documentation (for example, demonstrating intent or describing data restrictions/format.)

Research has show that even out-of-sync documentation can provide benefit. In particular, high-level descriptions of architecture and intent age far better than implementation specifics. Therefore, we can effectively avoid most of the threat of documentation drift by maintaining high-level documents, and by documenting primarily intent.

## 3. Where Avoidance Fails

If we would like to document an external API, avoidance does not provide a good strategy. For example, if we avoid documenting parameters in a publicly-exposed function, the user will have to read the source to understand what values to pass to a given function. This can be non-trivial if the parameters are tramp data (that is, if they are not used in the function being called, but are passed to some other function.)

## 4. Darglint

## 5. General Purpose

*Darglint* was written to provide an alternative to the avoidance strategy. Darglint helps prevent documentation drift by warning developers when docstrings do not match the actual function/method body.

### 5.1 Example

Let's say we have a base error class which defines a method, 'message'. A programmer recently decided that the class should not raise exceptions when run in production. To this end, they added a parameter, but forgot to document it:

```
def message(self, verbosity=1, raises=True):
    """Get the message for this error.

    Args:
        verbosity: An integer in the set {1,2},
            where 1 is a more terse message, and
            2 includes a general description.

    Raises:
        Exception: If the verbosity level is not
            recognized.

    Returns:
        An error message.

    """
    if verbosity == 1:
        return '{}'.format(self.terse_message)
    elif verbosity == 2:
        return '{}: {}'.format(
            self.general_message,
            self.terse_message,
        )
    else:
        if raises:
            raise Exception(
                'Unrecognized verbosity setting '
                + str(verbosity)
            )
```

Running *darglint*, below gives us a warning about the missing parameter:

```
> darglint -v 2 base_error.py
base_error.py:message:4: I101: Missing parameter(s)
    in Docstring: - raises
```

We now know that a parameter description was not updated with the method definition.

## 6. Features

*Darglint* includes many features which one would normally expect of a linter:

### 6.1 Error silencing

Errors can be silenced in two ways: using a configuration file, or by including a "noqa" statement in the docstring. For example, the below function would normally raise an exception because there is a missing "Returns" section. But we have prevented the error from occurring by including a "noqa". A bare "noqa" statement causes all errors to be ignored for the docstring.

```
def get_user_config():
    """Returns the configuration.

    # noqa: I201

    """
    from django.conf import settings
    return settings.USER_SETTINGS
```

### 6.2 Message templating

A template can be passed to *darglint* for formatting error messages. This can be useful for automatic processing. For example, the below call only prints the error number:

```
darglint --message-template "{msg_id}" *.py
```

### 6.3 Syntax errors

Errors will also be raised for problems with syntax. This makes it easier to ensure a consistent format and to make docstrings parsable for *darglint*.

### 6.4 Verbosity settings

There are two default verbosity settings more or less in-depth error messages. For example, the function

```
def insert(self, item):
    """Insert the given item into this binary tree."""
    ...
```

Will produce the following output from darglint:

```
> darglint -v 1 binary_tree.py
binary_tree.py:insert:11: I101: - item

> darglint -v 2 binary_tree.py
binary_tree.py:insert:11: I101: Missing parameter(s) in
    Docstring: - item
```

### 6.5 Type comparisons

When converting a project over to use type hints, it is easy for documentation to fall behind. If the docstring uses Google-doc style types, then it will compare them against the type signature and raise an error if they do not match. For example, the below function will raise the error "add_two.py:add_two:1: I103: y: expected int but was float".

```
def add_two(x: int, y: int) -> int:
    """Add two numbers.

    Args:
        x (int): The first number.
        y (float): The second number.

    Returns:
        int: The sum of the first and second number.

    """
    return x + y
```

## 7. Disadvantages

The primary disadvantage of *darglint* is that it only warns missing or extra parameters, return statments, etc. If you change how a parameter is used, or its order, then *darglint* is of no help.

## 8. Alternatives

There are many alternative strategies for mitigating documentation drift. One popular strategy for avoiding documentation drift is to extract documentation directly from the code. This is done, for example, by Swagger to generate descriptions of APIs using the OpenAPI specification. This, of course, is limited as it does not describe intent; it is a more convenient means of viewing the function definitions.

While avoidance is common, and *darglint* attempts to use control to prevent documentation drift, there are relatively few attempts at using assessment as a preventative/remedial measure. One possibility entertained by a commenter on Reddit (/u/robertmeta) is to use version control to see whether comments are likely to be out-of-date:

> It got so bad on a codebase I worked on a young dev who joined the team wrote a little git utility to check the function comment to function body changes and mark them all with date diffs and "absolute lie" "probably a lie" "maybe true" "probably true".

A more sophisticated utility could identify code which has been altered more recently than the documentation which describes it. You could then order all documentation in a code base by how out-of-sync it is likely to be. Such a Pareto chart would make a good starting point for documentation review.

A final idea is to tie documentation to some specific set of tests. This is the idea behind *literate_integration*, a small framework for writing integration tests designed to be run and generated into documentation. (The idea is based on the concept of literate programming from Donald Knuth.) In *literate_integration*, you write from which you can generate documentation in Markdown. If the test fails, then the documentation is out of date and must be updated. Because the documentation is strongly coupled with the test, updating the test makes it more likely for the documentation to be up to date.

## References


[1] C. J. Satish and M. Anand *Software Documentation Management Issues and Practices: a Survey* Indian Journal of Science and Technology Tamilnadu, India May 2016

[2] Timothy C. Lethbridge and Janice Singer and Andrew Forward *How Software Engineers Use Documentation: The State of the Practice* IEEE Software November/December 2003