Import Linter Documentation

Release 1.0b1

David Seddon

Jul 03, 2019

Contents

CHAPTER 1

Import Linter

Import Linter allows you to define and enforce rules for the internal and external imports within your Python project.

- Free software: BSD license
- Documentation: https://import-linter.readthedocs.io.

Warning: This software is currently in alpha. This means there are likely to be changes that break backward compatibility. However, due to it being a development tool (rather than something that needs to be installed on a production system), it may be suitable for inclusion in your testing pipeline. It also means we actively encourage people to try it out and submit bug reports.

1.1 Overview

Import Linter is a command line tool to check that you are following a self-imposed architecture within your Python project. It does this by analysing the imports between all the modules in a Python package, and compares this against a set of rules that you provide in a configuration file.

The configuration file contains one or more 'contracts'. Each contract has a specific type, which determines the sort of rules it will apply. For example, the `independence` contract type checks that there are no imports, in either direction, between a set of subpackages.

Import Linter is particularly useful if you are working on a complex codebase within a team, when you want to enforce a particular architectural style. In this case you can add Import Linter to your deployment pipeline, so that any code that does not follow the architecture will fail tests.

If there isn't a built in contract type that fits your desired architecture, you can define a custom one.

1.2 Quick start

Install Import Linter:

```
pip install import-linter
```

Decide on the dependency flows you wish to check. In this example, we have decided to make sure that there are no dependencies between `myproject.foo` and `myproject.bar`, so we will use the `independence` contract type.

Create an `.importlinter` file in the root of your project. For example:

```
[importlinter]
root_package = myproject

[importlinter:contract:1]
name=Foo and bar are decoupled
type=independence
modules=
    myproject.foo
    myproject.bar
```

Now, from your project root, run:

```
lint-imports
```

If your code violates the contract, you will see an error message something like this:

```
=============
Import Linter
=============

---------
Contracts
---------

Analyzed 23 files, 44 dependencies.
-----------------------------------

Foo and bar are decoupled BROKEN

Contracts: 1 broken.


----------------
Broken contracts
----------------

Foo and bar are decoupled
-------------------------

myproject.foo is not allowed to import myproject.bar:

-   myproject.foo.blue -> myproject.utils.red (l.16)
    myproject.utils.red -> myproject.utils.green (l.1)
    myproject.utils.green -> myproject.bar.yellow (l.3)
```

For more details, see *Usage*.

CHAPTER 2

Installation

At the command line:

```
pip install import-linter
```

CHAPTER 3

Usage

3.1 Configuration file location

Before running the linter, you need to supply configuration in a file, in INI format. Import Linter will look in the current directory for one of the following files:

- `setup.cfg`
- `.importlinter`

(Different filenames / locations can be specified as a command line argument, see below.)

3.2 Top level configuration

Your file must contain an `importlinter` section providing top-level (i.e. non-contract based) configuration:

```
[importlinter]
# Required:
root_package = mypackage
# Optional:
include_external_packages = True
```

Options:

- `root_package`: The name of the top-level Python package to validate. This package must be importable: usually this means it is has been installed using pip, or it's in the current directory. (Required.)
- `include_external_packages`: Whether to include external packages when building the import graph (see the Grimp build_graph documentation for more details). This is not currently used by any built in contracts, but it could be used by a custom contract type that wanted to enforce rules relating to packages not in the root package (i.e. in the Python standard library, or in third party libraries). (Optional.)

3.3 Contracts

Additionally, you will want to include one or more contract configurations. These take the following form:

```
[importlinter:contract:1]
name = Contract One
type = some_contract_type
(additional options)

[importlinter:contract:2]
name = Contract Two
type = another_contract_type
(additional options)
```

Notice each contract has its own INI section, which begins `importlinter:contract:` and ends in an arbitrary, unique code (in this example, the codes are `1` and `2`). These codes are purely to adhere to the INI format, which does not allow duplicate section names.

Every contract will always have the following key/value pairs:

- `name`: A human-readable name for the contract.
- `type`: The type of contract to use (see *Contract types*.)

Each contract type defines additional options that you supply here.

3.4 Running the linter

Import Linter provides a single command: `lint-imports`.

Running this will check that your project adheres to the contracts you've defined.

Arguments:

- `--config`: The configuration file to use. If not supplied, Import Linter will look for `setup.cfg` or `.importlinter` in the current directory. (Optional.)

Default usage:

```
lint-imports
```

Using a different filename or location:

```
lint-imports --config path/to/alternative-config.ini
```

CHAPTER 4

Contract types

4.1 Independence

Type name: `independence`

Independence contracts check that a set of modules do not depend on each other.

They do this by checking that there are no imports in any direction between the modules, even indirectly.

Example:

```
[importlinter:contract:1]
name = My independence contract
type = independence
modules =
    mypackage.foo
    mypackage.bar
    mypackage.baz
ignore_imports =
    mypackage.bar.green -> mypackage.utils
    mypackage.baz.blue -> mypackage.foo.purple
```

Configuration options

- `modules`: A list of modules/subpackages that should be independent from each other.
- `ignore_imports`: A list of imports, each in the form `mypackage.foo.importer -> mypackage.bar.imported`. These imports will be ignored: if the import would cause a contract to be broken, adding it to the list will cause the contract be kept instead. (Optional.)

4.2 Layers

Type name: `layers`

Layers contracts enforce a 'layered architecture', where higher layers may depend on lower layers, but not the other way around.

They do this by checking, for an ordered list of modules, that none higher up the list imports anything from a module lower down the list, even indirectly. To allow for a repeated pattern of layers across a project, you also define a set of 'containers', which are treated as the parent package of the layers.

Layers are required by default: if a layer is listed in the contract, the contract will be broken if the layer doesn't exist. You can make a layer optional by wrapping it in parentheses.

Examples

```
[importlinter:contract:1]
name = My three-tier layers contract
type = layers
layers=
    high
    medium
    low
containers=
    mypackage
```

This contract will not allow imports from lower layers to higher layers. For example, it will not allow `mypackage.low` to import `mypackage.high`, even indirectly.

```
[importlinter:contract:1]
name = My multiple package layers contract
type = layers
layers=
    high
    (medium)
    low
containers=
    mypackage.foo
    mypackage.bar
    mypackage.baz
```

In this example, each container has its own layered architecture. For example, it will not allow `mypackage.foo.low` to import `mypackage.foo.high`. However, it will allow `mypackage.foo.low` to import `mypackage.bar.high`, as they are in different containers:

Notice that `medium` is an optional layer. This means that if it is missing from any of the containers, Import Linter won't complain.

Configuration options

- `layers`: An ordered list with the name of each layer module, *relative to its parent package*. The order is from higher to lower level layers.
- `containers`: List of the parent modules of the layers, as *absolute names* that you could import, such as `mypackage.foo`. If you only have one set of layers, there will only be one container.
- `ignore_imports`: A list of imports, each in the form `mypackage.foo.importer -> mypackage.bar.imported`. These imports will be ignored: if the import would cause a contract to be broken, adding it to the list will cause the contract be kept instead. (Optional.)

4.3 Custom contract types

If none of the built in contract types meets your needs, you can define a custom contract type: see *Custom contract types*.

CHAPTER 5

Custom contract types

If none of the built in contract types serve your needs, you can define a custom contract type. The steps to do this are:

1. Somewhere in your Python path, create a module that implements a `Contract` class for your supplied type.
2. Register the contract type in your configuration file.
3. Define one or more contracts of your custom type, also in your configuration file.

5.1 Step one: implementing a Contract class

You define a custom contract type by subclassing `importlinter.Contract` and implementing the following methods:

- **`check(graph):`** Given an import graph of your project, return a `ContractCheck` describing whether the contract was adhered to.

  Arguments:

  – `graph`: a Grimp `ImportGraph` of your project, which can be used to inspect / analyse any dependencies. For full details of how to use this, see the Grimp documentation.

  Returns:

  – An `importlinter.ContractCheck` instance. This is a simple dataclass with two attributes, `kept` (a boolean indicating if the contract was kept) and `metadata` (a dictionary of data about the check). The metadata can contain anything you want, as it is only used in the `render_broken_contract` method that you also define in this class.

- `render_broken_contract(check):`

  Renders the results of a broken contract check. For output, this should use the `importlinter.output` module.

  Arguments:

  – `check`: the `ContractCheck` instance returned by the `check` method above.

Contract fields

A contract will usually need some further configuration. This can be done using *fields*. For an example, see `importlinter.contracts.layers`.

Example custom contract

```python
from importlinter import Contract, ContractCheck, fields, output


class ForbiddenImportContract(Contract):
    """
    Contract that defines a single forbidden import between
    two modules.
    """
    importer = fields.StringField()
    imported = fields.StringField()

    def check(self, graph):
        forbidden_import_details = graph.get_import_details(
            importer=self.importer,
            imported=self.imported,
        )
        import_exists = bool(forbidden_import_details)

        return ContractCheck(
            kept=not import_exists,
            metadata={
                'forbidden_import_details': forbidden_import_details,
            }
        )

    def render_broken_contract(self, check):
        output.print_error(
            f'{self.importer} is not allowed to import {self.imported}:',
            bold=True,
        )
        output.new_line()
        for details in check.metadata['forbidden_import_details']:
            line_number = details['line_number']
            line_contents = details['line_contents']
            output.indent_cursor()
            output.print_error(f'{self.importer}:{line_number}: {line_contents}')
```

5.2 Step two: register the contract type

In the `[importlinter]` section of your configuration file, include a list of `contract_types` that map type names onto the Python path of your custom class:

```ini
[importlinter]
root_package_name = mypackage
contract_types =
    forbidden_import: somepackage.contracts.ForbiddenImportContract
```

5.3 Step three: define your contracts

You may now use the type name defined in the previous step to define a contract:

```
[importlinter:contract:1]
name = My custom contract
type = forbidden_import
importer = mypackage.foo
imported = mypackage.bar
```

CHAPTER 6

Contributing

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

6.1 Bug reports

When 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.

6.2 Documentation improvements

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

6.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/seddonym/import-linter/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 code contributions are welcome :)

6.4 Development

To set up *import-linter* for local development:

1. Fork import-linter (look for the "Fork" button).
2. Clone your fork locally:

   ```
   git clone git@github.com:your_name_here/import-linter.git
   ```

3. Create a branch for local development:

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

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

   ```
   tox
   ```

5. 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
   ```

6. Submit a pull request through the GitHub website.

6.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

6.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though . . .

CHAPTER 7

Authors

- David Seddon - http://seddonym.me

CHAPTER 8

Changelog

8.1 1.0a1 (2019-1-27)

- Release blank project on PyPI.

8.2 1.0a2 (2019-3-26)

- First usable alpha release.

8.3 1.0a3 (2019-3-27)

- Include the ability to build the graph with external packages.

8.4 1.0b1 (2019-4-6)

- Improve error handling of modules/containers not in the graph.
- Return the exit code correctly.
- Run lint-imports on Import Linter itself.
- Allow single values in ListField.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search