Import Linter Documentation

Release 1.7.0

David Seddon

Jan 27, 2023

Contents

CHAPTER 1

Import Linter

Import Linter allows you to define and enforce rules for the imports within and between Python packages.

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

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 one or more Python packages, 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 `forbidden` contract type allows you to check that certain modules or packages are not imported by parts of your project.

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 `myproject.foo` has dependencies on neither `myproject.bar` nor `myproject.baz`, so we will use the `forbidden` contract type.

Create an `.importlinter` file in the root of your project to define your contract(s). In this case:

```
[importlinter]
root_package = myproject

[importlinter:contract:1]
name=Foo doesn't import bar or baz
type=forbidden
source_modules=
    myproject.foo
forbidden_modules=
    myproject.bar
    myproject.baz
```

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 doesn't import bar or baz BROKEN

Contracts: 1 broken.


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

Foo doesn't import bar or baz
-----------------------------

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. If not specified over the command line, Import Linter will look in the current directory for one of the following files:

- `setup.cfg` (INI format)
- `.importlinter` (INI format)
- `pyproject.toml` (TOML format)

3.2 Top level configuration

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

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

Or, with multiple root packages:

```
[importlinter]
root_packages=
    packageone
    packagetwo
# Optional:
include_external_packages = True
```

Options:

- `root_package`: The name of the Python package to validate. For regular packages, this must be the top level package (i.e. one with no dots in its name). However, in the special case of namespace packages, the name of the

portion should be supplied, for example `'mynamespace.foo'`. This package must be importable: usually this means it is has been installed using pip, or it's in the current directory. (Either this or `root_packages` is required.)

- `root_packages`: The names of the Python packages to validate. This should be used in place of `root_package` if you want to analyse the imports of multiple packages, and is subject to the same requirements. (Either this or `root_package` is required.)
- `include_external_packages`: Whether to include external packages when building the import graph. Unlike root packages, external packages are *not* statically analyzed, so no imports from external packages will be checked. However, imports *of* external packages will be available for checking. Not every contract type uses this. For more information, see the Grimp build_graph documentation. (Optional.)

3.3 Contracts

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

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

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

Notice each contract has its own INI section, which begins `importlinter:contract:` and ends in a unique id (in this example, the ids are `one` and `two`). These codes can be used to to select individual contracts when running the linter (see below).

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. This overrides the default file search strategy. By default it's assumed that the file is an ini-file unless the file extension is `toml`. (Optional.)
- `--contract`: Limit the check to the contract with the supplied id. In INI files, a contract's id is the final part of the section header: for example, the id for a contract with a section header of `[importlinter:contract:foo]` is `foo`. In TOML files, ids are supplied explicitly with an `id` key. This option may be provided multiple times to check more than one contract. (Optional.)
- `--show_timings`: Display the times taken to build the graph and check each contract. (Optional.)
- `--verbose`: Noisily output progress as it goes along. (Optional.)

Default usage:

```
lint-imports
```

Using a different filename or location:

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

Checking only certain contracts:

```
lint-imports --contract some-contract --contract another-contract
```

Showing timings:

```
lint-imports --show-timings
```

Verbose mode:

```
lint-imports --verbose
```

CHAPTER 4

Python API

While it is usually run via the command line, Import Linter offers a Python API for certain functions.

4.1 Reading configuration

```
>>> from importlinter import api
>>> api.read_configuration()
{
    "session_options": {"root_packages": ["importlinter"]},
    "contracts_options": [
        {
            "containers": ["importlinter"],
            "layers": [
                "cli",
                "api",
                "configuration",
                "adapters",
                "contracts",
                "application",
                "domain",
            ],
            "name": "Layered architecture",
            "type": "layers",
        }
    ],
}
```

read_configuration(*config_filename=None*)

Return a dictionary containing configuration from the supplied file.

If no filename is supplied, look in the default location (see *Usage*).

This function is designed for use by external projects wishing to analyse the contracts themselves, e.g. to track the number of ignored imports.

Parameters **config_filename** (*str*) – The path to the file containing the configuration (optional).

Returns

A dictionary with two keys:

- `"session_options"`: dictionary of strings passed as top level configuration. Note that if a single `root_package` is in the configuration, it will be normalised to a single-item list of `root_packages`, as shown in the example above.
- `"contracts_options"`: list of dictionaries, one for each contract, keyed with:
  - `"name"`: the name of the contract (str).
  - `"type"`: the type of the contract (str).
  - Any other contract-specific configuration.

Return type dict

CHAPTER 5

Contract types

5.1 Forbidden modules

Type name: `forbidden`

Forbidden contracts check that one set of modules are not imported by another set of modules.

Descendants of each module will be checked - so if `mypackage.one` is forbidden from importing `mypackage.two`, then `mypackage.one.blue` will be forbidden from importing `mypackage.two.green`. Indirect imports will also be checked.

External packages may also be forbidden.

Examples:

```
[importlinter]
root_package = mypackage

[importlinter:contract:my-forbidden-contract]
name = My forbidden contract (internal packages only)
type = forbidden
source_modules =
    mypackage.one
    mypackage.two
    mypackage.three.blue
forbidden_modules =
    mypackage.four
    mypackage.five.green
ignore_imports =
    mypackage.one.green -> mypackage.utils
    mypackage.two -> mypackage.four
```

```
[importlinter]
root_package = mypackage
include_external_packages = True
```

(continues on next page)

(continued from previous page)

```
[importlinter:contract:my-forbidden-contract]
name = My forbidden contract (internal and external packages)
type = forbidden
source_modules =
    mypackage.one
    mypackage.two
forbidden_modules =
    mypackage.three
    django
    requests
ignore_imports =
    mypackage.one.green -> sqlalchemy
```

Configuration options

Configuration options:

- `source_modules`: A list of modules that should not import the forbidden modules.
- `forbidden_modules`: A list of modules that should not be imported by the source modules. These may include root level external packages (i.e. `django`, but not `django.db.models`). If external packages are included, the top level configuration must have `internal_external_packages = True`.
- `ignore_imports`: See *Options used by multiple contracts*.
- `unmatched_ignore_imports_alerting`: See *Options used by multiple contracts*.
- `allow_indirect_imports`: If `True`, allow indirect imports to forbidden modules without interpreting them as a reason to mark the contract broken. (Optional.)

5.2 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:my-independence-contract]
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`: See *Options used by multiple contracts*.
- `unmatched_ignore_imports_alerting`: See *Options used by multiple contracts*.

5.3 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.

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.

You may also define a set of 'containers'. These allow for a repeated pattern of layers across a project. If containers are provided, these are treated as the parent package of the layers.

If you want to make sure that *every* module in each container is defined as a layer, you can mark the contract as 'exhaustive'. This means that if a module is added to the code base in the same package as your layers, the contract will fail. Any such modules that shouldn't cause a failure can be added to an `exhaustive_ignores` list. At present, exhaustive contracts are only supported for layers that define containers.

Examples

```
[importlinter]
root_package = mypackage

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

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]
root_packages=
    high
    medium
    low

[importlinter:contract:my-layers-contract]
name = My three-tier layers contract (multiple root packages)
type = layers
layers=
    high
    medium
    low
```

This contract is similar to the one above, but is suitable if the packages are not contained within a root package (i.e. the Python project consists of several packages in a directory that does not contain an `__init__.py` file). In this case, `high`, `medium` and `low` all need to be specified as `root_packages` in the `[importlinter]` configuration.

```
[importlinter:contract:my-layers-contract]
name = My multiple package layers contract
type = layers
```

(continues on next page)

(continued from previous page)

```
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.

This is an example of an 'exhaustive' contract.

```
[importlinter:contract:my-layers-contract]
name = My multiple package layers contract
type = layers
layers=
    high
    (medium)
    low
containers=
    mypackage.foo
    mypackage.bar
    mypackage.baz
exhaustive = true
exhaustive_ignores =
    utils
```

If, say, a module existed called `mypackage.foo.extra`, the contract will fail as it is not listed as a layer. However `mypackage.foo.utils` would be allowed as it is listed in `exhaustive_ignores`.

Configuration options

- `layers`: An ordered list with the name of each layer module. If containers are specified, then these names must be *relative to the container*. The order is from higher to lower level layers. Layers wrapped in parentheses (e.g. `(foo)`) will be ignored if they are not present in the file system.
- `containers`: List of the parent modules of the layers, as *absolute names* that you could import, such as `mypackage.foo`. (Optional.)
- `ignore_imports`: See *Options used by multiple contracts*.
- `unmatched_ignore_imports_alerting`: See *Options used by multiple contracts*.
- `exhaustive`. If true, check that the contract declares every possible layer in its list of layers to check. (Optional, default False.)
- `exhaustive_ignores`. A list of layers to ignore in exhaustiveness checks. (Optional.)

5.4 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*.

5.5 Options used by multiple contracts

- `ignore_imports`: Optional 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.

  Wildcards (in the form of `*`) are supported. These can stand in for a module names, but they do not extend to subpackages.

  Examples:

  - `mypackage.*`: matches `mypackage.foo` but not `mypackage.foo.bar`.
  - `mypackage.*.baz`: matches `mypackage.foo.baz` but not `mypackage.foo.bar.baz`.
  - `mypackage.*.*`: matches `mypackage.foo.bar` and `mypackage.foobar.baz`.
  - `mypackage.foo*`: not a valid expression. (The wildcard must replace a whole module name.)

- `unmatched_ignore_imports_alerting`: The alerting level for handling expressions supplied in `ignore_imports` that do not match any imports in the graph. Choices are:

  - `error`: Error if there are any unmatched expressions (default).
  - `warn`: Print a warning for each unmatched expression.
  - `none`: Do not alert.

CHAPTER 6

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.

6.1 Step one: implementing a Contract class

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

- **`check(graph: ImportGraph, verbose: bool) -> ContractCheck:`** 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.
  - `verbose`: Whether we're in *verbose mode*. You can use this flag to determine whether to output text during the check, using `output.verbose_print`, as in the example below.

  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: ContractCheck) -> None:`

  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

```
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, verbose):
        output.verbose_print(
            verbose,
            f"Getting import details from {self.importer} to {self.imported}..."
        )
        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}')
```

6.2 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:

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

6.3 Step three: define your contracts

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

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

CHAPTER 7

TOML support

While all the examples are in INI format, Import Linter also supports TOML.

The TOML configuration is very similar to the others with a few differences:

- the sections must start with `tool.`
- contracts are defined by `[[tool.importlinter.contracts]]`

The basic configuration layout looks like:

```
[tool.importlinter]
root_package = mypackage

[[tool.importlinter.contracts]]
name = Contract One

[[tool.importlinter.contracts]]
name = Contract Two
```

Following, an example with a layered configuration:

```
[tool.importlinter]
root_packages = [
    "high",
    "medium",
    "low",
]

[[tool.importlinter.contracts]]
name = "My three-tier layers contract (multiple root packages)"
type = "layers"
layers = [
    "high",
    "medium",
    "low",
]
```

7.1 Contract ids

You can optionally provide an `id` key for each contract. This allows you to make use of the `--contract` parameter when *running the linter*.

CHAPTER 8

Contributing

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

8.1 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.

8.2 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.

8.3 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 :)

8.4 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.

8.4.1 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`.

8.4.2 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 Github Actions - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 9

Authors

- David Seddon - https://seddonym.me

CHAPTER 10

Contributors

- Anthony Sottile - https://github.com/asottile
- Łukasz Skarżyński - https://github.com/skarzi
- Daniel Jurczak - https://github.com/danieljurczak
- Ben Warren - https://github.com/bwarren
- Aaron Gokaslan - https://github.com/Skylion007
- Kai Mueller - https://github.com/kasium
- Daniele Esposti - https://github.com/expobrain
- Petter Friberg - https://github.com/flaeppe
- James Owen - https://github.com/leamingrad
- Matthew Gamble - https://github.com/mwgamble

CHAPTER 11

Changelog

11.1 1.7.0 (2023-01-27)

- Switch from optional dependency of `toml` to required dependency of `tomli` for Python versions < 3.11.
- Use DetailedImport type hinting made available in Grimp 2.2.
- Allow limiting by contract.

11.2 1.6.0 (2022-12-7)

- Add exhaustiveness option to layers contracts.

11.3 1.5.0 (2022-12-2)

- Officially support Python 3.11.

11.4 1.4.0 (2022-10-04)

- Include py.typed file in package data to support type checking
- Remove upper bounds on dependencies. This allows usage of Grimp 2.0, which should significantly speed up checking of layers contracts.
- Add –verbose flag to lint-imports command.
- **Improve algorithm for independence contracts, in the following ways:**
  - It is significantly faster.

  - As with layers contracts, reports of illegal indirect imports reports now include multiple start and end points (if they exist).
  - Illegal indirect imports that are via other modules listed in the contract are no longer listed.

11.5 1.3.0 (2022-08-22)

- Add Python API for reading configuration.
- Add support for namespace packages.

11.6 1.2.7 (2022-04-04)

- Officially support Python 3.10.
- Drop support for Python 3.6.
- Add support for default Field values.
- Add EnumField.
- Support warnings in contract checks.
- Add unmatched_ignore_imports_alerting option for each contract.
- Add command line argument for showing timings.

11.7 1.2.6 (2021-09-24)

- Fix bug with ignoring external imports that occur multiple times in the same module.

11.8 1.2.5 (2021-09-21)

- Wildcard support for ignored imports.
- Convert TOML booleans to strings in UserOptions, to make consistent with INI file parsing.

11.9 1.2.4 (2021-08-09)

- Fix TOML installation bug.

11.10 1.2.3 (2021-07-29)

- Add support for TOML configuration files.

11.11 1.2.2 (2021-07-13)

- Support Click version 8.

11.12 1.2.1 (2021-01-22)

- Add allow_indirect_imports to Forbidden Contract type
- Upgrade Grimp to 1.2.3.
- Officially support Python 3.9.

11.13 1.2 (2020-09-23)

- Upgrade Grimp to 1.2.2.
- Add SetField.
- Use a SetField for ignore_imports options.
- Add support for non *w* characters in import exceptions.

11.14 1.1 (2020-06-29)

- Bring 1.1 out of beta.

11.15 1.1b2 (2019-11-27)

- Update to Grimp v1.2, significantly increasing speed of building the graph.

11.16 1.1b1 (2019-11-24)

- Provide debug mode.
- Allow contracts to mutate the graph without affecting other contracts.
- Update to Grimp v1.1.
- Change the rendering of broken layers contracts by combining any shared chain beginning or endings.
- Speed up and make more comprehensive the algorithm for finding illegal chains in layer contracts. Prior to this, layers contracts used Grimp's find_shortest_chains method for each pairing of layers. This found the shortest chain between each pair of modules across the two layers. The algorithm was very slow and not comprehensive. With this release, for each pair of layers, a copy of the graph is made. All other layers are removed from the graph, any direct imports between the two layers are stored. Next, the two layers in question are 'squashed', the shortest chain is repeatedly popped from the graph until no more chains remain. This results in more comprehensive results, and at significantly increased speed.

11.17 1.0 (2019-17-10)

- Officially support Python 3.8.

11.18 1.0b5 (2019-10-05)

- Allow multiple root packages.
- Make containers optional in Layers contracts.

11.19 1.0b4 (2019-07-03)

- Add https://pre-commit.com configuration.
- Use find_shortest_chains instead of find_shortest_chain on the Grimp import graph.
- Add Forbidden Modules contract type.

11.20 1.0b3 (2019-05-15)

- Update to Grimp v1.0b10, fixing Windows incompatibility.

11.21 1.0b2 (2019-04-16)

- Update to Grimp v1.0b9, fixing error with using importlib.util.find_spec.

11.22 1.0b1 (2019-04-06)

- 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.

11.23 1.0a3 (2019-03-27)

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

11.24 1.0a2 (2019-03-26)

- First usable alpha release.

11.25 1.0a1 (2019-01-27)

- Release blank project on PyPI.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Index

R