Layer Linter Documentation

Release 0.12.3

David Seddon

Jun 08, 2019

Contents:

1 Outline **3**

2 Overview **5**

3 Quick start **7**

3.1 Installation 8
3.2 Usage 9
3.3 Core concepts 10
3.4 Contributing 14
3.5 Credits 14
3.6 History 15
3.7 Migrating to Import Linter 17

Layer Linter has been deprecated in favour of Import Linter.

Import Linter does everything Layer Linter does, but with more features and a slightly different API. If you're already using Layer Linter, migrating to Import Linter is simple: there is a guide here.

CHAPTER 1

Outline

Layer Linter checks that your project follows a custom-defined layered architecture, based on its internal dependencies (i.e. the imports between its modules).

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

CHAPTER 2

Overview

Layer 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 internal imports between all the modules in your code base, and compares this against a set of simple rules that you provide in a `layers.yml` file.

For example, you can use it to check that no modules inside `myproject.foo` import from any modules inside `myproject.bar`, even indirectly.

This 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 Layer Linter to your deployment pipeline, so that any code that does not follow the architecture will fail tests.

CHAPTER 3

Quick start

Install Layer Linter:

```
pip install layer-linter
```

Decide on the dependency flows you wish to check. In this example, we have organised our project into three subpackages, `myproject.high`, `myproject.medium` and `myproject.low`. These subpackages are known as *layers*. Note: layers must have the same parent package (i.e. all be in the same directory). This parent is known as a *container*.

Create a `layers.yml` in the root of your project. For example:

```
My Layers Contract:
  containers:
    - myproject
  layers:
    - high
    - medium
    - low
```

(This contract tells Layer Linter that the order of the layers runs from `low` at the bottom to `high` at the top. Layers higher up can import ones lower down, but not the other way around.)

Note that the container is an absolute name of a Python package, while the layers are relative to the container.

Now, from your project root, run:

```
layer-lint myproject
```

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

```
============
Layer Linter
============

---------
```

(continues on next page)

(continued from previous page)

```
Contracts
---------

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

My layer contract BROKEN

Contracts: 0 kept, 1 broken.

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


My layer contract
-----------------


1. myproject.low.x imports myproject.high.y:

    myproject.low.x <-
    myproject.utils <-
    myproject.high.y
```

For more details, see Usage.

3.1 Installation

3.1.1 Requirements

Layer Linter currently only supports Python 3.6 - 3.7.

3.1.2 Stable release

To install Layer Linter, run this command in your terminal:

```
$ pip install layer-linter
```

This is the preferred method to install Layer Linter, 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.

3.1.3 Development version

The sources for Layer Linter can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/seddonym/layer_linter
```

Or download the tarball:

```
$ curl  -OL https://github.com/seddonym/layer_linter/tarball/master
```

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

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

3.2 Usage

Before use, you will probably want to read *Core concepts*.

3.2.1 Defining your contracts

Your layers contracts are defined in a YAML file named `layers.yml`. This may exist anywhere, but a good place is in your project root.

The file contains one or more contracts, in the following format:

```
Contract name:
    containers:
        - mypackage.container
        ...
    layers:
        - layerone
        - layertwo
        ...
    whitelisted_paths:
        - mypackage.container.layertwo.module <- mypackage.container.layerone.module
        ...
```

1. **Contract name**: A string to describe your contract.
2. **Containers**: Absolute names of any Python package that contains the layers as immediate children. One or more containers are allowed in this list.
3. **Layers**: Names of the Python module *relative* to each container listed in `containers`. Modules lower down the list must not import modules higher up. (Remember, a Python module can either be a `.py` file or a directory with an `__init__.py` file inside.)
4. **Whitelisted paths** (optional): If you wish certain import paths not to break in the contract, you can optionally whitelist them. The modules should be listed as absolute names, with the importing module first, and the imported module second.

For some examples, see *Core concepts*.

3.2.2 Additional syntax

Optional layers

You may specify certain layers as optional, by enclosing the name in parentheses.

By default, Layer Linter will error if it cannot find the module for a particular layer. Take the following contract:

```
My contract:
    containers:
        - mypackage.foo
        - mypackage.bar
    layers:
        - one
        - two
```

If the module `mypackage.foo.two` is missing, the contract will be broken. If you want the contract to pass despite this, you can enclose the layer name in parentheses:

```
My contract:
    containers:
        - mypackage.foo
        - mypackage.bar
    layers:
        - one
        - (two)
```

Layer `two` is now optional, which means the contract will pass even though `mypackage.bar.two` is missing.

3.2.3 Running the linter

Layer Linter provides a single command: `layer-lint`.

Running this will check that your project adheres to the contracts in your `layers.yml`.

- Positional arguments:
  - `package_name`: The name of the top-level Python package to validate (required).
- Optional arguments:
  - `--config`: The YAML file describing your layer contract(s). If not supplied, Layer Linter will look for a file called `layers.yml` in the current directory.
  - `--quiet`: Do not output anything if the contracts are all adhered to.
  - `--verbose` (or `-v`): Output a more verbose report.
  - `--debug`: Output debug messages when running the linter. No parameters required.

Default usage:;

```
layer-lint myproject
```

Using a different filename or location instead of `layers.yml`:

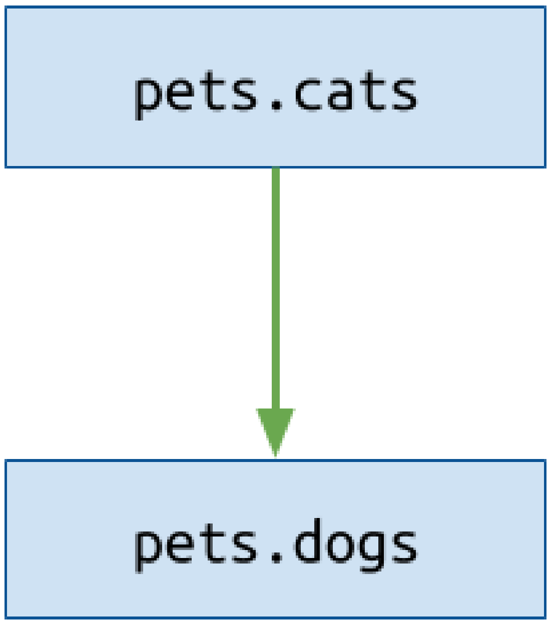
```
layer-lint myproject --config path/to/alternative.yml
```

3.3 Core concepts

3.3.1 The Dependency Graph

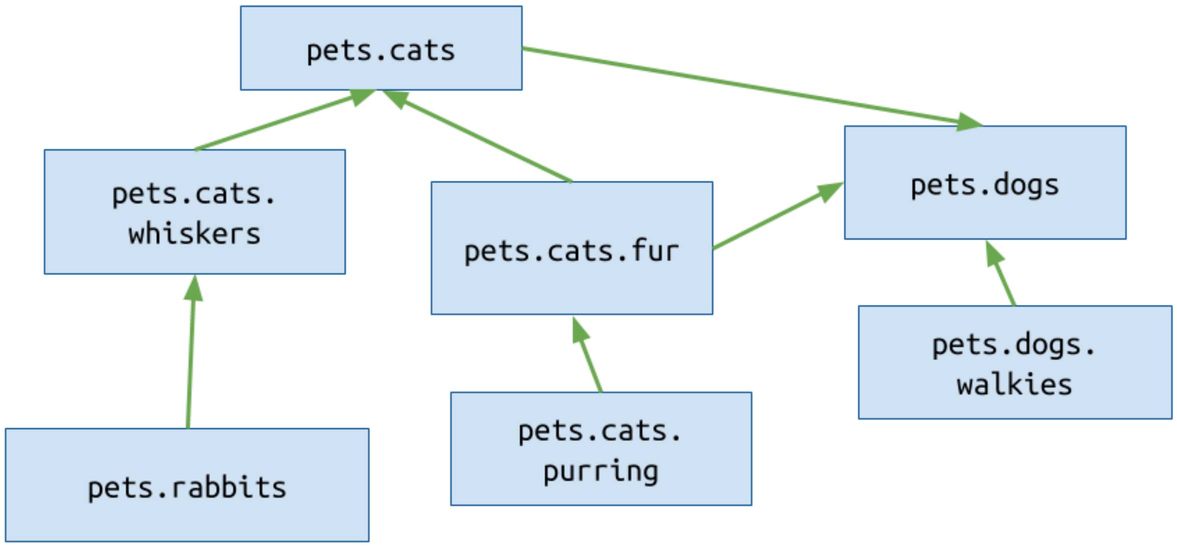
At the heart of Layer Linter is a graph of internal dependencies within a Python code base. This is a graph in a mathematical sense: a collection of items with relationships between them. In this case, the items are Python modules, and the relationships are imports between them.

For example, a project named `pets` with two modules, where `pets.dogs` imports `pets.cats`, would have a graph like this:



Note the direction of the arrow, which we'll use throughout: the arrow points from the imported module into the importing module.

If the project was larger, it would have a more complex graph:



When you run Layer Linter, it statically analyses all of your code to produce a graph like this. (Note: these are just visual representations of the underlying data structure; Layer Linter has no visual output.)

3.3.2 Layers

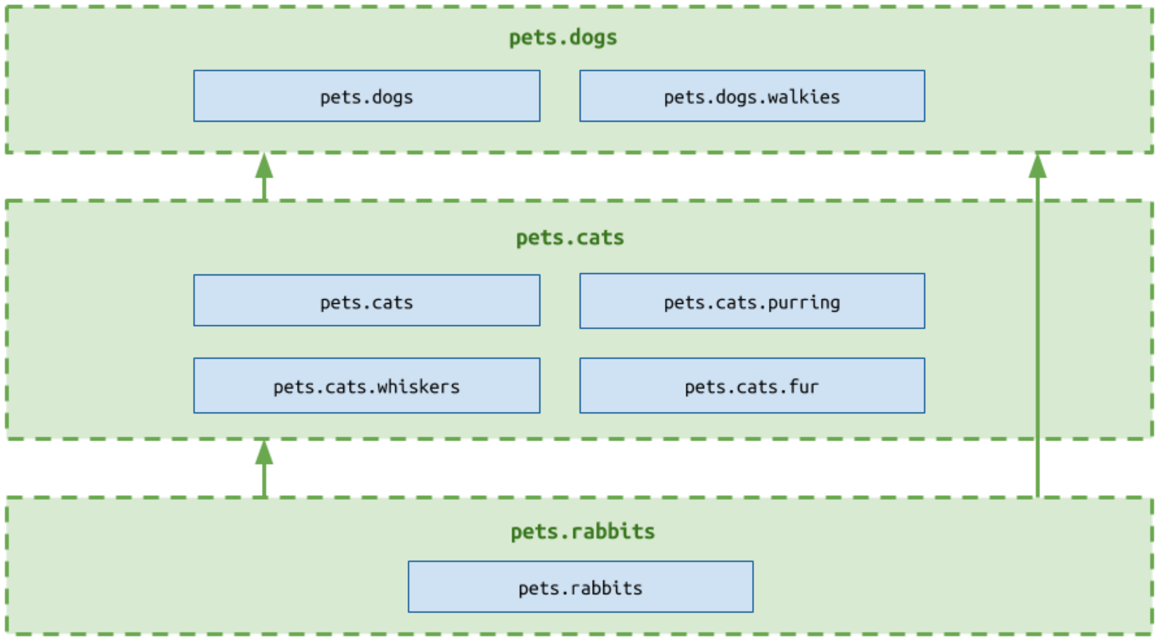
Layers are a concept used in software architecture. They describe an application organized into distinct sections, or *layers*.

In such an architecture, lower layers should be ignorant of higher ones. This means that code in a higher layer can use utilities provided in a lower layer, but not the other way around. In other words, there is a dependency flow from low to high.

Layers in Python

In Python, you can think of a layer as a single `.py` file, or a package containing multiple `.py` files. This layer is grouped with other layers, all sharing a common parent package: in other words, a group of layers will all be in the same directory, at the same level. Layer Linter calls this common parent a *container*.

Within a single group of layers, any file within a higher up layer may import from any file lower down, but not the other way around - even indirectly.



The above example shows a single group consisting of three layers. Their container is the top level package, `pets`. According to the constraints imposed by layers, `pets.cats.purring` may import `pets.rabbits` but not `pets.dogs.walkies`. `pets.dogs.walkies` may import any other module, as it is in the highest layer.

(For further reading on Layers, see the Wikipedia page on Multitier Architecture).

3.3.3 Contracts

Contracts are how you describe your architecture to Layer Linter. You write them in a `layers.yml` file. Each Contract contains two lists, `layers` and `containers`.

- `layers` takes the form of an ordered list with the name of each layer module, *relative to its parent package*. The order is from high level layer to low level layer.
- `containers` lists the parent modules of the layers, as *absolute names* that you could import, such as `mypackage.foo`. If you have only one set of layers, there will be only one container: the top level pack-

age. However, you could choose to have a repeating pattern of layers across multiple subpackages; in which case, you would include each of those subpackages in the containers list.

You can have as many of these contracts as you like, and you give each one a name.

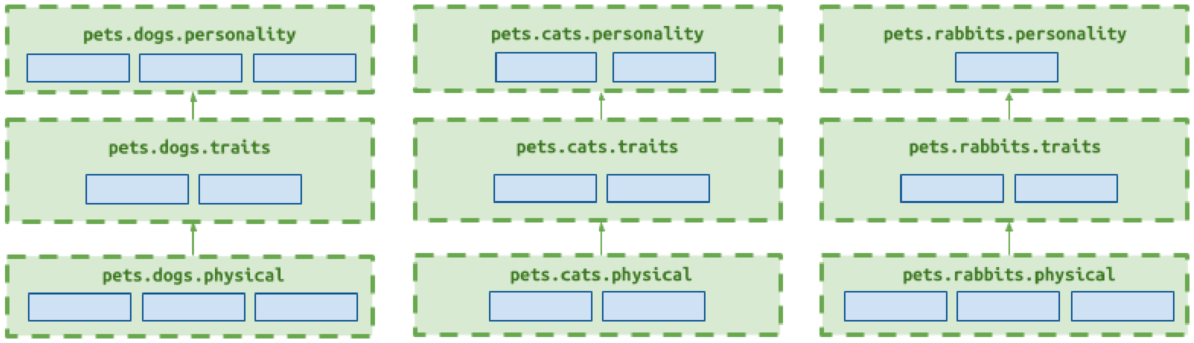
Example: single container contract

The three-layered structure described earlier can be described by the following contract. Note that the layers have names relative to the single, containing package.

```
Three-tier contract:
    containers:
        - pets
    layers:
        - dogs
        - cats
        - rabbits
```

Example: multiple package contract

A more complex architecture might involve the same layers repeated across multiple containers, like this:



In this case, rather than have three contracts, one for each container, you may list all the containers in a single contract. The order of the containers is not important.

```
Modular contract:
    containers:
        - pets.dogs
        - pets.cats
        - pets.rabbits
    layers:
        - personality
        - traits
        - physical
```

3.3.4 Whitelisting paths

Sometimes, you may wish to tolerate certain dependencies that do not adhere to your contract. To do this, include them as *whitelisted paths* in your contract.

Let's say you have a project that has a `utils` module that introduces an illegal dependency between two of your layers. The report might look something like this:

```
----------------
Broken contracts
----------------


My layer contract
-----------------

1. pets.cats.whiskers imports pets.dogs.walkies:

    pets.cats.whiskers <-
    pets.utils <-
    pets.dogs.walkies
```

To suppress this error, you may add one component of the path to the contract like so:

```
Three-tier contract:
    containers:
        - pets
    layers:
        - dogs
        - cats
        - rabbits
    whitelisted_paths:
        - pets.cats.whiskers <- pets.utils
```

Running the linter again will show the contract passing.

There are a few use cases:

- Your project does not completely adhere to the contract, but you want to prevent it getting worse. You can whitelist any known issues, and gradually fix them.
- You have an exceptional circumstance in your project that you are comfortable with, and don't wish to fix.
- You want to understand how many dependencies you would need to fix before a project conforms to a particular architecture. Because Layer Linter only shows the most direct dependency violation, whitelisting paths can reveal less direct ones.

3.4 Contributing

This package has now been superseded by Import Linter, so we are not considering new features here. If you wish to contribute, please get involved with that package.

3.5 Credits

3.5.1 Development Lead

- David Seddon <david@seddonym.me>

3.5.2 Contributors

- Thiago Colares https://github.com/colares

- James Cooke https://github.com/jamescooke
- Simon Biggs https://github.com/SimonBiggs

3.5.3 Other credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

3.6 History

3.6.1 0.1.0 (2018-06-20)

- First release on PyPI.

3.6.2 0.2.0 (2018-06-23)

- Look for `layers.yml` in current working directory.

3.6.3 0.3.0 (2018-06-24)

- Renamed command to `layer-lint`.
- Changed order of layers in `layers.yml` to be listed high level to low level.

3.6.4 0.4.0 (2018-07-22)

- Made dependency analysis more efficient and robust.
- Improved report formatting.
- Removed illegal dependencies that were implied by other, more succinct illegal dependencies.
- Added `--debug` command line argument.

3.6.5 0.5.0 (2018-08-01)

- Added count of analysed files and dependencies to report.
- Fixed issues from running command in a different directory to the package.
- Increased speed of analysis.
- Changed `--config_directory` command line argument to `--config-directory`.

3.6.6 0.6.0 (2018-08-07)

- Added ability to whitelist paths.

3.6.7 0.6.1 (2018-08-07)

- Added current working directory to path.

3.6.8 0.6.2 (2018-08-17)

- Don't analyse children of directories that aren't Python packages.
- Prevented installing incompatible version of Pydeps (1.6).

3.6.9 0.7.0 (2018-09-04)

- Completed rewrite of static analysis used to build dependency graph.
- Added quiet and verbose reporting.
- Added type annotation and mypy.
- Built earlier versions of Python using pybackwards.
- Corrected docs to refer to `layers.yml` instead of `layers.yaml`.

3.6.10 0.7.1 (2018-09-04)

- Fixed packaging bug with 0.7.0.

3.6.11 0.7.2 (2018-09-05)

- Fixed bug with not checking all submodules of layer.

3.6.12 0.7.3 (2018-09-07)

- Dropped support for Python 3.4 and 3.5 and adjust packaging.

3.6.13 0.7.4 (2018-09-20)

- Tweaked command line error handling.
- Improved README and *Core Concepts* documentation.

3.6.14 0.8.0 (2018-09-29)

- Replace `--config-directory` parameter with `--config` parameter, which takes a file name instead.

3.6.15 0.9.0 (2018-10-13)

- Moved to beta version.
- Improved documentation.
- Better handling of invalid package names passed to command line.

3.6.16 0.10.0 (2018-10-14)

- Renamed ‘packages’ to ‘containers’ in contracts.

3.6.17 0.10.1 (2018-10-14)

- Improved handling of invalid containers.

3.6.18 0.10.2 (2018-10-17)

- Error if a layer is missing.

3.6.19 0.10.3 (2018-11-2)

- Fixed RST rendering on PyPI.

3.6.20 0.11.0 (2018-11-5)

- Support defining optional layers.

3.6.21 0.11.1 (2019-1-16)

- Updated dependencies, especially switching to a version of PyYAML to address https://nvd.nist.gov/vuln/detail/CVE-2017-18342.

3.6.22 0.12.0 (2019-1-16)

- Fix parsing of relative imports within __init__.py files.

3.6.23 0.12.1 (2019-2-2)

- Add support for Click 7.x.

3.6.24 0.12.2 (2019-3-20)

- Fix bug with Windows file paths.

3.6.25 0.12.3 (2019-6-8)

- Deprecate Layer Linter in favour of Import Linter.

3.7 Migrating to Import Linter

If you would like to migrate from Layer Linter to Import Linter, you can easily migrate your setup.

3.7.1 Step One - Install

```
pip install import-linter
```

3.7.2 Step Two - Configure

Import Linter uses INI instead of YAML to define its contracts. Create an `.importlinter` file in the same directory as your `layers.yml`. This is where your configuration will live.

Converting the format is simple.

Example `layers.yml`:

```yaml
Contract one:
    containers:
        - mypackage
    layers:
        - api
        - utils

Contract two:
    # This is a comment.
    containers:
        - mypackage.foo
        - mypackage.bar
        - mypackage.baz
    layers:
        - top
        - middle
        - bottom
    whitelisted_paths:
        - mypackage.foo.bottom.alpha <- mypackage.foo.middle.beta
```

Equivalent `.importlinter`:

```ini
[importlinter]
root_package = mypackage


[importlinter:contract:1]
name = Contract one
type = layers
containers=
    mypackage
layers=
    api
    utils


[importlinter:contract:2]
# This is a comment.
name = Contract two
type = layers
containers=
    mypackage.foo
    mypackage.bar
```

(continues on next page)

(continued from previous page)

```
    mypackage.baz
layers=
    top
    middle
    bottom
ignore_imports=
    mypackage.foo.bottom.alpha -> mypackage.foo.middle.beta
```

Things to note:

- Import Linter requires the root package to be configured in the file, rather than passed to the command line.
- Each contract requires a `type` - this is because Import Linter supports other contract types.
- Each contract needs an arbitrary unique identifier in the INI section (in this case, `1` and `2`).
- 'Whitelisted paths' has have been renamed to 'ignore imports'. The notation is similar except the arrow is reversed; the importing package is still listed first, the imported package second.

3.7.3 Step Three - Run

To lint your package, run:

```
lint-imports
```

Or, if your configuration file is in a different directory:

```
lint-imports --config=path/to/.importlinter
```

3.7.4 Key differences between the packages

- You may notice slight differences in the imports Import Linter picks up on. The main example is that it does not ignore modules in `migrations` subpackages, while Layer Linter does.
- Import Linter allows you to use other contract types and even define your own.
- Import Linter allows you to analyse imports of external packages too (though these don't make sense in the context of a layers contract).

Further reading can be found in the Import Linter documentation.