
Grimp Documentation

Release 1.0b12

David Seddon

Jun 12, 2019

Contents

1	Grimp	1
1.1	Quick start	1
2	Installation	3
3	Usage	5
3.1	Terminology	5
3.2	Building the graph	6
3.3	Methods for analysing the module tree	6
3.4	Methods for analysing direct imports	7
3.5	Methods for analysing import chains	8
3.6	Methods for manipulating the graph	9
4	Contributing	11
4.1	Bug reports	11
4.2	Documentation improvements	11
4.3	Feature requests and feedback	11
4.4	Development	12
5	Authors	13
6	Changelog	15
6.1	0.0.1 (2018-11-05)	15
6.2	1.0b1 (2018-12-08)	15
6.3	1.0b2 (2018-12-12)	15
6.4	1.0b3 (2018-12-16)	15
6.5	1.0b4 (2019-1-7)	15
6.6	1.0b5 (2019-1-12)	16
6.7	1.0b6 (2019-1-20)	16
6.8	1.0b7 (2019-1-21)	16
6.9	1.0b8 (2019-2-1)	16
6.10	1.0b9 (2019-4-16)	16
6.11	1.0b10 (2019-5-15)	16
6.12	1.0b11 (2019-5-18)	16
6.13	1.0b12 (2019-6-12)	16
7	Indices and tables	17

CHAPTER 1

Grimp

Builds a graph of a Python project's internal and external dependencies.

- Free software: BSD license

Warning: This software is currently in beta. It is undergoing active development, and breaking changes may be introduced between versions.

1.1 Quick start

Install grimp:

```
pip install grimp
```

Install the Python package you wish to analyse:

```
pip install somepackage
```

In Python, build the import graph for the package:

```
>>> import grimp
>>> graph = grimp.build_graph('somepackage')
```

You may now use the graph object to analyse the package. Some examples:

```
>>> graph.find_children('somepackage.foo')
{
    'somepackage.foo.one',
    'somepackage.foo.two',
}
```

(continues on next page)

(continued from previous page)

```
}

>>> graph.find_descendants('somepackage.foo')
{
    'somepackage.foo.one',
    'somepackage.foo.two',
    'somepackage.foo.two.blue',
    'somepackage.foo.two.green',
}

>>> graph.find_modules_directly_imported_by('somepackage.foo')
{
    'somepackage.bar.one',
}

>>> graph.find_upstream_modules('somepackage.foo')
{
    'somepackage.bar.one',
    'somepackage.baz',
    'somepackage.foobar',
}

>>> graph.find_shortest_chain(importer='somepackage.foobar', imported='somepackage.foo
↳')
(
    'somepackage.foobar',
    'somepackage.baz',
    'somepackage.foo',
)

>>> graph.get_import_details(importer='somepackage.foobar', imported='somepackage.baz
↳')
[
    {
        'importer': 'somepackage.foobar',
        'imported': 'somepackage.baz',
        'line_number': 5,
        'line_contents': 'from . import baz',
    },
]
```

By default, external dependencies will not be included. This can be overridden like so:

```
>>> graph = grimp.build_graph('somepackage', include_external_packages=True)
>>> graph.find_modules_directly_imported_by('somepackage.foo')
{
    'somepackage.bar.one',
    'os',
    'decimal',
    'sqlalchemy',
}
```

For the full list of methods, see *Usage*.

CHAPTER 2

Installation

At the command line:

```
pip install grimp
```


Grimp provides an API in the form of an `ImportGraph` that represents all the imports within a top-level Python package. This object has various methods that make it easy to find out information about that package's structure and interdependencies.

3.1 Terminology

The terminology around Python packages and modules can be a little confusing. Here are the definitions we use, taken in part from [the official Python docs](#):

- **Module:** A file containing Python definitions and statements. This includes ordinary `.py` files and `__init__.py` files.
- **Package:** A special kind of module that namespaces other modules using dotted module names. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Packages take the form of `__init__.py` files in a container directory. Packages may contain other packages. *A package is also a module.*
- **Top Level Package:** A package in the root namespace - in other words, one that is not a subpackage. For example, `A` is a top level package, but `A.B` is not.
- **Graph:** A graph in the [mathematical sense](#) of a collection of items with relationships between them. Grimp's `ImportGraph` is a directed graph of imports contained in a particular top level package.
- **Direct Import:** An import from one module to another.
- **Import Chain:** A chain of direct imports between two modules, possibly via other modules. For example, if `mypackage.foo` imports `mypackage.bar`, which in turn imports `mypackage.baz`, then there is an import chain between `mypackage.foo` and `mypackage.baz`.
- **Squashed Module:** A module in the graph that represents both itself and all its descendants. Squashed modules allow parts of the graph to be simplified. For example, if you include external packages when building the graph, each external package will exist in the graph as a single squashed module.

3.2 Building the graph

```
import grimp

graph = grimp.build_graph('mypackage')
```

`grimp.build_graph(package_name, include_external_packages=False)`
Build and return an ImportGraph for the supplied package.

Parameters

- **package_name** (*str*) – The name of the top level package, for example 'mypackage'.
- **include_external_packages** (*bool*) – Whether to include external packages in the import graph. If this is True, any other top level packages that are imported by this top level package (including packages in the standard library) will be included in the graph as squashed modules (see [Terminology](#) above). Note: external packages are only analysed as modules that are imported; any imports they make themselves will not be included in the graph.

Returns An import graph that you can use to analyse the package.

Return type ImportGraph

3.3 Methods for analysing the module tree

ImportGraph.modules

All the modules contained in the graph.

return Set of module names.

rtype A set of strings.

ImportGraph.find_children(module)

Return all the immediate children of the module, i.e. the modules that have a dotted module name that is one level below.

param str module The importable name of the module, e.g. 'mypackage' or 'mypackage.foo.one'. This may be any module within the package. It doesn't need to be a package itself, though if it isn't, it will have no children.

return Set of module names.

rtype A set of strings.

raises ValueError if the module is a squashed module, as by definition it represents both itself and all of its descendants.

ImportGraph.find_descendants(module)

Return all the descendants of the module, i.e. the modules that have a dotted module name that is below the supplied module, to any depth.

param str module The importable name of the module, e.g. 'mypackage' or 'mypackage.foo.one'. As with `find_children`, this doesn't have to be a package, though if it isn't then the set will be empty.

return Set of module names.

rtype A set of strings.

raises `ValueError` if the module is a squashed module, as by definition it represents both itself and all of its descendants.

3.4 Methods for analysing direct imports

`ImportGraph.direct_import_exists(importer, imported, as_packages=False)`

Parameters

- **importer** (*str*) – A module name.
- **imported** (*str*) – A module name.
- **as_packages** (*bool*) – Whether or not to treat the supplied modules as individual modules, or as entire packages (including any descendants).

Returns Whether or not the importer directly imports the imported module.

Return type `True` or `False`.

`ImportGraph.find_modules_directly_imported_by(module)`

Parameters **module** (*str*) – A module name.

Returns Set of all modules in the graph are imported by the supplied module.

Return type A set of strings.

`ImportGraph.find_modules_that_directly_import(module)`

Parameters **module** (*str*) – A module name.

Returns Set of all modules in the graph that directly import the supplied module.

Return type A set of strings.

`ImportGraph.get_import_details(importer, imported)`

Provides a way of seeing the details of direct imports between two modules (usually there will be only one of these, but it is possible for a module to import another module twice).

The details are in the following form:

```
[
  {
    'importer': 'mypackage.importer',
    'imported': 'mypackage.imported',
    'line_number': 5,
    'line_contents': 'from mypackage import imported',
  },
  # (additional imports here)
]
```

Parameters

- **importer** (*str*) – A module name.
- **imported** (*str*) – A module name.

Returns A list of the details of every direct import between two modules.

Return type List of dictionaries.

`ImportGraph.count_imports()`

Returns The number of direct imports in the graph.

Return type Integer.

3.5 Methods for analysing import chains

`ImportGraph.find_downstream_modules(module, as_package=False)`

Parameters

- **module** (*str*) – A module name.
- **as_package** (*bool*) – Whether or not to treat the supplied module as an individual module, or as an entire package (including any descendants). If treating it as a package, the result will include downstream modules *external* to the supplied module, and won't include modules within it.

Returns All the modules that import (even indirectly) the supplied module.

Return type A set of strings.

Examples:

```
# Returns the modules downstream of mypackage.foo.
import_graph.find_downstream_modules('mypackage.foo')

# Returns the modules downstream of mypackage.foo, mypackage.foo.one and
# mypackage.foo.two.
import_graph.find_downstream_modules('mypackage.foo', as_package=True)
```

`ImportGraph.find_upstream_modules(module, as_package=False)`

Parameters

- **module** (*str*) – A module name.
- **as_package** (*bool*) – Whether or not to treat the supplied module as an individual module, or as a package (i.e. including any descendants, if there are any). If treating it as a subpackage, the result will include upstream modules *external* to the package, and won't include modules within it.

Returns All the modules that are imported (even indirectly) by the supplied module.

Return type A set of strings.

`ImportGraph.find_shortest_chain(importer, imported)`

Parameters

- **importer** (*str*) – The module at the start of a potential chain of imports between `importer` and `imported` (i.e. the module that potentially imports `imported`, even indirectly).
- **imported** (*str*) – The module at the end of the potential chain of imports.

Returns The shortest chain of imports between the supplied modules, or `None` if no chain exists.

Return type A tuple of strings, ordered from importer to imported modules, or `None`.

`ImportGraph.find_shortest_chains(importer, imported)`

Parameters

- **importer** (*str*) – A module or subpackage within the graph.
- **imported** (*str*) – Another module or subpackage within the graph.

Returns The shortest import chains that exist between the `importer` and `imported`, and between any modules contained within them. Only one chain per upstream/downstream pair will be included. Any chains that are contained within other chains in the result set will be excluded.

Return type A set of tuples of strings. Each tuple is ordered from `importer` to `imported` modules.

`ImportGraph.chain_exists(importer, imported, as_packages=False)`

Parameters

- **importer** (*str*) – The module at the start of the potential chain of imports (as in `find_shortest_chain`).
- **imported** (*str*) – The module at the end of the potential chain of imports (as in `find_shortest_chain`).
- **as_packages** (*bool*) – Whether to treat the supplied modules as individual modules, or as packages (including any descendants, if there are any). If treating them as packages, all descendants of `importer` and `imported` will be checked too.

Returns Return whether any chain of imports exists between `importer` and `imported`, even indirectly; in other words, does `importer` depend on `imported`?

Return type `bool`

3.6 Methods for manipulating the graph

`ImportGraph.add_module(module, is_squashed=False)`

Add a module to the graph.

Parameters

- **module** (*str*) – The name of a module, for example `'mypackage.foo'`.
- **is_squashed** (*bool*) – If `True`, the module should be treated as a ‘squashed module’ (see [Terminology](#) above).

Returns `None`

`ImportGraph.remove_module(module)`

Remove a module from the graph.

If the module is not present in the graph, no exception will be raised.

Parameters **module** (*str*) – The name of a module, for example `'mypackage.foo'`.

Returns `None`

`ImportGraph.add_import(importer, imported, line_number=None, line_contents=None)`

Add a direct import between two modules to the graph. If the modules are not already present, they will be added to the graph.

Parameters

- **importer** (*str*) – The name of the module that is importing the other module.
- **imported** (*str*) – The name of the module being imported.
- **line_number** (*int*) – The line number of the import statement in the module.

- **line_contents** (*str*) – The line that contains the import statement.

Returns None

`ImportGraph.remove_import (importer, imported)`

Remove a direct import between two modules. Does not remove the modules themselves.

Parameters

- **importer** (*str*) – The name of the module that is importing the other module.
- **imported** (*str*) – The name of the module being imported.

Returns None

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

4.1 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.

4.2 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.

4.3 Feature requests and feedback

The best way to send feedback is to file an issue at <https://github.com/seddonym/grimp/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 :)

4.4 Development

To set up *grimp* for local development:

1. Fork [grimp](#) (look for the “Fork” button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/grimp.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.

4.4.1 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`.

4.4.2 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 5

Authors

- David Seddon - <http://seddonym.me>

6.1 0.0.1 (2018-11-05)

- Release blank project on PyPI.

6.2 1.0b1 (2018-12-08)

- Implement core functionality.

6.3 1.0b2 (2018-12-12)

- Fix PyPI readme rendering.

6.4 1.0b3 (2018-12-16)

- Fix bug with analysing relative imports from within `__init__.py` files.
- Stop skipping analysing packages called `migrations`.
- Deal with invalid imports by warning instead of raising an exception.
- Rename `NetworkXBackedImportGraph` to `ImportGraph`.

6.5 1.0b4 (2019-1-7)

- Improve repr of `ImportGraph`.
- Fix bug with `find_shortest_path` using upstream/downstream the wrong way around.

6.6 1.0b5 (2019-1-12)

- Rename `get_shortest_path` to `get_shortest_chain`.
- Rename `path_exists` to `chain_exists`.
- Rename and reorder the kwargs for `get_shortest_chain` and `chain_exists`.
- Raise `ValueError` if modules with shared descendants are passed to `chain_exists` if `as_packages=True`.

6.7 1.0b6 (2019-1-20)

- Support building the graph with external packages.

6.8 1.0b7 (2019-1-21)

- Add `count_imports` method.

6.9 1.0b8 (2019-2-1)

- Add `as_packages` parameter to `direct_import_exists`.

6.10 1.0b9 (2019-4-16)

- Fix bug with calling `importlib.util.find_spec`.

6.11 1.0b10 (2019-5-15)

- Fix Windows incompatibility.

6.12 1.0b11 (2019-5-18)

- Add `remove_module` method.

6.13 1.0b12 (2019-6-12)

- Add `find_shortest_chains` method.

CHAPTER 7

Indices and tables

- `genindex`
- `modindex`
- `search`

G

`grimp.build_graph()` (*built-in function*), 6

I

`ImportGraph.add_import()` (*built-in function*), 9

`ImportGraph.add_module()` (*built-in function*), 9

`ImportGraph.chain_exists()` (*built-in function*), 9

`ImportGraph.count_imports()` (*built-in function*), 7

`ImportGraph.direct_import_exists()`
(*built-in function*), 7

`ImportGraph.find_children()` (*built-in function*), 6

`ImportGraph.find_descendants()` (*built-in function*), 6

`ImportGraph.find_downstream_modules()`
(*built-in function*), 8

`ImportGraph.find_modules_directly_imported_by()`
(*built-in function*), 7

`ImportGraph.find_modules_that_directly_import()`
(*built-in function*), 7

`ImportGraph.find_shortest_chain()` (*built-in function*), 8

`ImportGraph.find_shortest_chains()`
(*built-in function*), 8

`ImportGraph.find_upstream_modules()`
(*built-in function*), 8

`ImportGraph.get_import_details()` (*built-in function*), 7

`ImportGraph.remove_import()` (*built-in function*), 10

`ImportGraph.remove_module()` (*built-in function*), 9

M

`modules` (*ImportGraph attribute*), 6