Grimp Documentation

Release 2.2

David Seddon

Jan 05, 2023

Contents

CHAPTER 1

Grimp

Builds a queryable graph of the imports within one or more Python packages.

- Free software: BSD license

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',
    },
]
```

1.2 External packages

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',
}
```

1.3 Multiple packages

You may analyse multiple root packages. To do this, pass each package name as a positional argument:

```
>>> graph = grimp.build_graph('somepackage', 'anotherpackage')
>>> graph.find_modules_directly_imported_by('somepackage.foo')
{
    'somepackage.bar.one',
    'anotherpackage.baz',
}
```

1.4 Namespace packages

Graphs can also be built from portions of namespace packages. To do this, provide the portion name, rather than the namespace name:

```
>>> graph = grimp.build_graph('somenamespace.foo')
```

1.4.1 What's a namespace package?

Namespace packages are a Python feature allows subpackages to be distributed independently, while still importable under a shared namespace. This is, for example, used by the Python client for Google's Cloud Logging API. When installed, it is importable in Python as `google.cloud.logging`. The parent packages `google` and `google.cloud` are both namespace packages, while `google.cloud.logging` is known as the 'portion'. Other portions in the same namespace can be installed separately, for example `google.cloud.secretmanager`.

Grimp expects the package name passed to `build_graph` to be a portion, rather than a namespace package. So in the case of the example above, the graph should be built like so:

```
>>> graph = grimp.build_graph('google.cloud.logging')
```

If, instead, a namespace package is passed (e.g. `grimp.build_graph('google.cloud')`), Grimp will raise `NamespacePackageEncountered`.

CHAPTER 2

Installation

At the command line:

```
pip install grimp
```

CHAPTER 3

Usage

Grimp provides an API in the form of an `ImportGraph` that represents all the imports within one or more top-level Python packages. This object has various methods that make it easy to find out information about the packages' structures 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 Python module which can contain submodules or recursively, subpackages.
- **Top Level Package**: A package that is not a subpackage of another package.
- **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 between modules.
- **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

# Single package
graph = grimp.build_graph('mypackage')

# Multiple packages
graph = grimp.build_graph('mypackage', 'anotherpackage', 'onemore')

# Include imports of external packages
graph = grimp.build_graph('mypackage', include_external_packages=True)
```

grimp.**build_graph**(*package_name*, **additional_package_names*, *include_external_packages=False*)

Build and return an ImportGraph for the supplied package or packages.

Parameters

- **package_name** (*str*) – The name of an importable package, for example `'mypackage'`. 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'`.
- **..] additional_package_names** (*tuple[str,]*) – Tuple of any additional package names. These can be supplied as positional arguments, as in the example above.
- **include_external_packages** (*bool*) – Whether to include external packages in the import graph. If this is `True`, any other top level packages (including packages in the standard library) that are imported by this package will be included in the graph as squashed modules (see *Terminology* above).

  The behaviour is more complex if one of the internal packages is a namespace portion. In this case, the squashed module will have the shallowest name that doesn't clash with any internal modules. For example, in a graph with internal packages `namespace.foo` and `namespace.bar.one.green`, `namespace.bar.one.orange.alpha` would be added to the graph as `namespace.bar.one.orange`. However, in a graph with only `namespace.foo` as an internal package, the same external module would be added as `namespace.bar`.

  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 a module in the graph, e.g. `'mypackage'` or `'mypackage.foo.one'`. This may be any non-squashed module. 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 any available metadata about direct imports between two modules. Usually the list will consist of a single dictionary, but it is possible for a module to import another module more than once.

This method should not be used to determine whether an import is present: some of the imports in the graph may have no available metadata. For example, if an import has been added by the `add_import` method without the `line_number` and `line_contents` specified, then calling this method on the import will return an empty list. If you want to know whether the import is present, use `direct_import_exists`.

The details returned are in the following form:

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

If no such import exists, or if there are no available details, an empty list will be returned.

Parameters

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

Returns A list of any available metadata for imports between two modules.

Return type List of dictionaries with the structure shown above. If you want to use type annotations, you may use the `grimp.DetailedImport` TypedDict for each dictionary.

`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

ImportGraph.**squash_module**(*module*)
'Squash' a module in the graph (see *Terminology* above).

Squashing a pre-existing module will cause all imports to and from the descendants of that module to instead point directly at the module being squashed. The import details (i.e. line numbers and contents) will be lost for those imports. The descendants will then be removed from the graph.

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

Returns None

ImportGraph.**is_module_squashed**(*module*)
Return whether a module present in the graph is 'squashed' (see *Terminology* above).

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

Returns bool

CHAPTER 4

NetworkX

If you want to analyze the graph in a way that isn't provided by Grimp, you may want to consider converting the graph to a NetworkX graph.

NetworkX is a third-party Python library with a large number of algorithms for working with graphs.

4.1 Converting the Grimp graph to a NetworkX graph

First, you should install NetworkX (e.g. `pip install networkx`).

You can then build up a NetworkX graph as shown:

```
import grimp
import networkx

grimp_graph = grimp.build_graph("mypackage")

# Build a NetworkX graph from the Grimp graph.
networkx_graph = networkx.DiGraph()
for module in grimp_graph.modules:
    networkx_graph.add_node(module)
    for imported in grimp_graph.find_modules_directly_imported_by(module):
        networkx_graph.add_edge(module, imported)
```

CHAPTER 5

Contributing

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

5.1 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.

5.2 Documentation improvements

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

5.3 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 :)

5.4 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.

5.4.1 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`.

5.4.2 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, which will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 6

Authors

- David Seddon - https://seddonym.me
- Kevin Amado - https://github.com/kamadorueda
- Matthew Gamble - https://github.com/mwgamble
- NetworkX developers - The shortest path algorithm was adapted from the NetworkX library https://networkx.org/.

CHAPTER 7

Changelog

7.1 2.2 (2023-1-5)

- Annotate get_import_details return value with a DetailedImport.

7.2 2.1 (2022-12-2)

- Officially support Python 3.11.

7.3 2.0 (2022-9-27)

- Significantly speed up graph copying.
- Remove find_all_simple_chains method.
- No longer use a networkx graph internally.
- Fix bug where import details remained stored in the graph after removing modules or imports.

7.4 1.3 (2022-8-15)

- Officially support Python 3.9 and 3.10.
- Drop support for Python 3.6.
- Support namespaced packages.

7.5 1.2.3 (2021-1-19)

- Raise custom exception (NamespacePackageEncountered) if code under analysis appears to be a namespace package.

7.6 1.2.2 (2020-6-29)

- Raise custom exception (SourceSyntaxError) if code under analysis contains syntax error.

7.7 1.2.1 (2020-3-16)

- Better handling of source code containing non-ascii compatible characters

7.8 1.2 (2019-11-27)

- Significantly increase the speed of building the graph.

7.9 1.1 (2019-11-18)

- Clarify behaviour of get_import_details.
- Add module_is_squashed method.
- Add squash_module method.
- Add find_all_simple_chains method.

7.10 1.0 (2019-10-17)

- Officially support Python 3.8.

7.11 1.0b13 (2019-9-25)

- Support multiple root packages.

7.12 1.0b12 (2019-6-12)

- Add find_shortest_chains method.

7.13 1.0b11 (2019-5-18)

- Add remove_module method.

7.14 1.0b10 (2019-5-15)

- Fix Windows incompatibility.

7.15 1.0b9 (2019-4-16)

- Fix bug with calling importlib.util.find_spec.

7.16 1.0b8 (2019-2-1)

- Add as_packages parameter to direct_import_exists.

7.17 1.0b7 (2019-1-21)

- Add count_imports method.

7.18 1.0b6 (2019-1-20)

- Support building the graph with external packages.

7.19 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.

7.20 1.0b4 (2019-1-7)

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

7.21 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.

7.22 1.0b2 (2018-12-12)

- Fix PyPI readme rendering.

7.23 1.0b1 (2018-12-08)

- Implement core functionality.

7.24 0.0.1 (2018-11-05)

- Release blank project on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Index