Grimp Documentation

Release 1.0b3

David Seddon

Contents

Grimp

Builds a graph of a Python project's internal 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',
}

>>> 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_path(upstream_module='somepackage.foobar',
                             downstream_module='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',
    },
]
```

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

CHAPTER 2

Installation

At the command line:

```
pip install grimp
```

Usage

Grimp provides an API in the form of an `ImportGraph` that represents all the internal 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 all the internal imports contained in a particular top level package.

- **Direct Import**: An import from one module to another.

- **Import Path**: A chain of 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 path between `mypackage.foo` and `mypackage.baz`.

3.2 Building the graph

```
import grimp

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

grimp.**build_graph**(*package_name*)

>Build and return an ImportGraph for the supplied package.

>>**param str package_name** The name of the top level package, for example `'mypackage'`.

>>**return** An import graph that you can use to analyse the package.

>>**rtype** ImportGraph

3.3 Methods for analysing the module tree

ImportGraph.**modules**

>All the modules in the top level package.

>>**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.

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.

3.4 Methods for analysing direct imports

ImportGraph.**direct_import_exists**(*importer*, *imported*)

>**Parameters**

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

> **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.

3.5 Methods for analysing import paths

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_path`**(*upstream_module*, *downstream_module*)

> **Parameters**
>
> - **upstream_module** (`str`) – The module at the start of the potential import path (i.e. that the downstream module will import).
>
> - **downstream_module** (`str`) – The module at the end of the potential import path (i.e. that will import the upstream module).
>
> **Returns** The shortest import path from the upstream to the downstream module, if one exists, or an empty tuple if not.
>
> **Return type** A tuple of strings, ordered from upstream to downstream modules.

`ImportGraph.`**`path_exists`**(*upstream_module*, *downstream_module*, *as_packages=False*)

> **Parameters**
>
> - **upstream_module** (`str`) – The module at the start of the potential import path (i.e. that the downstream module will import).
>
> - **downstream_module** (`str`) – The module at the end of the potential import path (i.e. that will import the upstream module).
>
> - **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 the upstream and downstream modules will be checked too.
>
> **Returns** Return whether any import path exists between the upstream and the downstream module, even indirectly; in other words, does the downstream module depend on the upstream module?
>
> **Return type** bool

3.6 Methods for manipulating the graph

`ImportGraph.`**`add_module`**(*module*)
> Add a module to the graph.

Contributing

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 . . .

Authors

- David Seddon - http://seddonym.me

Changelog

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.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

G

I

M