modulegraph Documentation

Release 0.19.4

Ronald Oussoren

May 31, 2023

Contents

modulegraph determines a dependency graph between Python modules primarily by bytecode analysis for import statements.

modulegraph uses similar methods to `modulefinder` from the standard library, but uses a more flexible internal representation, has more extensive knowledge of special cases, and is extensible.

Contents:

CHAPTER 1

Release history

1.1 0.19.4

- Fix broken python 2.7 support

  PR by Josua Root
- Initial support for Python 3.12

  The changes to support Python 3.12 are a bit rough, tests pass but I'm not 100% convinced the changes are correct.

1.2 0.19.3

- Fix incompatibility with Python 3.11

1.3 0.19.2

- Fix project links in PyPI sidebar

1.4 0.19.1

- Explictly mention python 3.10 support in package metadata

1.5 0.19

- Fix incorrect path when package `__init__` is an extension.

1.6 0.18

- Avoid exception when one of the items on "packages" is not a package (modulegraph.find_modules.find_needed_modules)
- #45: `Modulegraph.foldReferences()` called the wrong method

  Reported by Anthony Foglia.

1.7 0.17

- The .pyc format changed a little in Python 3.7

1.8 0.16

Features:

- Add LICENSE file to distribution
- Don't rely on pkg_resources to calculate package version
- Replace use of `optparse` by `argparse` as the former is deprecated

  Patch by htgoebel
- Attempt to reduce the maximum recursion needed to create the ModuleGraph
- Don't include the file type in the result from `zipio.getmode`
- Fix mismatched indents/dedents in ModuleGraph debug output

  Patch by codewarrior0

Bug fixes:

- Testsuite now passes on Windows (testd through appveyor)

  This only required changes to remove platform dependencies from the test suite.

1.9 0.15

Features:

- Issue #39: Traceback with for a syntax error when compiling async function

  On Python 3.5 some (invalid) async function definitions caused a modulegraph traceback, instead of adding "InvalidSourceModule" nodes to the graph.
- Issue #40: The graph now contains nodes of type "InvalidRelativeImport" for attempts to use relative imports that walk outside of a toplevel package.
- Module `modulegraph.find_modules` can no longer be used a script, use `python -m modulegraph` instead.

Bugfixes:

- Issue 38: Dot output broken in Python 3

  Patch by user elnuno on bitbucket.

- Issue 36: Make sure test suite works on systems other than macOS

  Patch by Hartmut Goebel
- Add support for "async def" to the AST scanner, needed to properly recognize imports in async function definitions.

1.10 0.14

Bugfixes:

- #33: Error scanning bytecode on python 3.4 or later

  The code using `dis.get_instructions` to scan the bytecode on Python 3.4 or later didn't work properly causing problems when trying to scan bytecode.

1.11 0.13

- Various documentation fixes by Thomas Kluyver.
- Fix incompatibility with recent versions of setuptools

  See also issue #206 in py2apps tracker for more information.
- Python 3: Ignore BOM at start of input files when compiling them.

  This matches the behavior of CPython, and avoids hard to diagnose problems. See also issue #178 in the py2app tracker
- Python 3.6 introduced a new bytecode format (wordcode), adjust the bytecode scanner for that.

1.12 0.12.1

- Issue #25: Complex python files could cause an "maximum recursion depth exceeded" exception due to using stack-based recursion to walk the module AST.

1.13 0.12

- Added 'modulegraph.modulegraph.InvalidSourceModule'. This graph node is used for Python source modules that cannot be compiled (for example because they contain syntax errors).

  This is primarily useful for being able to create a graph for packages that have python 2.x or python 3.x compatibility in separate modules that contain code that isn't valid in the "other" python version.
- Added 'modulegraph.modulegraph.InvalidCompiledModule'. This graph node is used for Python bytecode modules that cannot be loaded.
- Added 'modulegraph.modulegraph.NamespacePackage'.

  Patch by bitbucket user htgoebel.
- No longer add a MissingModule node to the graph for 'collections.defaultdict' when using 'from collections import defaultdict' ('collections.defaultdict' is an attribute of 'collections', not a submodule).
- Fixed typo in ModuleGraph.getReferences()

- Added ModuleGraph.getReferers(tonode). This methods yields the nodes that are referencing *tonode* (the reverse of getReferences)
- The graph will no longer contain MissingModule nodes when using 'from … import name' to import a global variable in a python module.

  There will still be MissingModule nodes for global variables in C extentions, and for 'from missing import name' when 'missing' is itself a MissingModule.
- Issue #18: Don't assume that a PEP 302 loader object has a `path` attribute. That attribute is not documented and is not always present.

1.14 0.11.2

-

1.15 0.11.1

- Issue #145: Don't exclude the platform specific 'path' modules (like ntpath)

1.16 0.11

This is a feature release

1.16.1 Features

- Hardcode knowlegde about the compatibility aliases in the email module (for python 2.5 upto 3.0).

  This makes it possible to remove a heavy-handed recipe from py2app.
- Added `modegraph.zipio.getmode` to fetch the Unix file mode for a file.
- Added some handy methods to `modulegraph.modulegraph.ModuleGraph`.

1.17 0.10.5

This is a bugfix release

- Don't look at the file extension to determine the file type in modulegraph.find_modules.parse_mf_results, but use the class of the item.
- Issue #13: Improved handing of bad relative imports ("from .foo import bar"), these tended to raise confusing errors and are now handled like any other failed import.

1.18 0.10.4

This is a bugfix release

- There were no 'classifiers' in the package metadata due to a bug in setup.py.

1.19 0.10.3

This is a bugfix release

1.19.1 Bugfixes

- `modulegraph.find.modules.parse_mf_results` failed when the main script of a py2app module didn't have a file name ending in '.py'.

1.20 0.10.2

This is a bugfix release

1.20.1 Bugfixes

- Issue #12: modulegraph would sometimes find the wrong package *__init__* module due to using the wrong search method. One easy way to reproduce the problem was to have a toplevel module named *__init__*.

  Reported by Kentzo.

1.21 0.10.1

This is a bugfix release

1.21.1 Bugfixes

- Issue #11: creating xrefs and dotty graphs from modulegraphs (the –xref and –graph options of py2app) didn't work with python 3 due to use of APIs that aren't available in that version of python.

  Reported by Andrew Barnert.

1.22 0.10

This is a minor feature release

1.22.1 Features

- `modulegraph.find_modules.find_needed_modules` claimed to automaticly include subpackages for the "packages" argument as well, but that code didn't work at all.
- Issue #9: The modulegraph script is deprecated, use "python -mmodulegraph" instead.
- Issue #10: Ensure that the result of "zipio.open" can be used in a with statement (that is, `with zipio.open(...) as fp`.
- No longer use "2to3" to support Python 3.

  Because of this modulegraph now supports Python 2.6 and later.

- Slightly improved HTML output, which makes it easier to manipulate the generated HTML using JavaScript.

  Patch by anatoly techtonik.
- Ensure modulegraph works with changes introduced after Python 3.3b1.
- Implement support for PEP 420 ("Implicit namespace packages") in Python 3.3.
- `modulegraph.util.imp_walk` is deprecated and will be removed in the next release of this package.

1.22.2 Bugfixes

- The module graph was incomplete, and generated incorrect warnings along the way, when a subpackage contained import statements for submodules.

  An example of this is `sqlalchemy.util`, the `__init__.py` file for this package contains imports of modules in that modules using the classic relative import syntax (that is `import compat` to import `sqlalchemy.util.compat`). Until this release modulegraph searched the wrong path to locate these modules (and hence failed to find them).

1.23 0.9.2

This is a bugfix release

1.23.1 Bugfixes

- The 'packages' option to modulegraph.find_modules.find_modules ignored the search path argument but always used the default search path.
- The 'imp_find_modules' function in modulegraph.util has an argument 'path', this was a string in previous release and can now also be a sequence.
- Don't crash when a module on the 'includes' list doesn't exist, but warn just like for missing 'packages' (modulegraph.find_modules.find_modules)

1.24 0.9.1

This is a bugfix release

1.24.1 Bug fixes

- Fixed the name of nodes imports in packages where the first element of a dotted name can be found but the rest cannot. This used to create a MissingModule node for the dotted name in the global namespace instead of relative to the package.

  That is, given a package "pkg" with submodule "sub" if the "__init__.py" of "pkg" contains "import sub.nomod" we now create a MissingModule node for "pkg.sub.nomod" instead of "sub.nomod".

  This fixes an issue with including the crcmod package in application bundles, first reported on the pythonmac-sig mailinglist by Brendan Simon.

1.25 0.9

This is a minor feature release

Features:

- Documentation is now generated using sphinx and can be viewed at <http://packages.python.org/modulegraph>.

  The documention is very rough at this moment and in need of reorganisation and language cleanup. I've basiclly writting the current version by reading the code and documenting what it does, the order in which classes and methods are document is therefore not necessarily the most useful.

- The repository has moved to bitbucket

- Renamed `modulegraph.modulegraph.AddPackagePath` to `addPackagePath`, likewise `ReplacePackage` is now `replacePackage`. The old name is still available, but is deprecated and will be removed before the 1.0 release.

- `modulegraph.modulegraph` contains two node types that are unused and have unclear semantics: `FlatPackage` and `ArchiveModule`. These node types are deprecated and will be removed before 1.0 is released.

- Added a simple commandline tool (`modulegraph`) that will print information about the dependency graph of a script.

- Added a module (`zipio`) for dealing with paths that may refer to entries inside zipfiles (such as source paths referring to modules in zipped eggfiles).

  With this addition `modulegraph.modulegraph.os_listdir` is deprecated and it will be removed before the 1.0 release.

Bug fixes:

- The `__cmp__` method of a Node no longer causes an exception when the compared-to object is not a Node. Patch by Ivan Kozik.

- Issue #1: The initialiser for `modulegraph.ModuleGraph` caused an exception when an entry on the path (`sys.path`) doesn't actually exist.

  Fix by "skurylo", testcase by Ronald.

- The code no longer worked with python 2.5, this release fixes that.

- Due to the switch to mercurial setuptools will no longer include all required files. Fixed by adding a MANIFEST.in file

- The method for printing a `.dot` representation of a `ModuleGraph` works again.

1.26 0.8.1

This is a minor feature release

Features:

- `from __future__ import absolute_import` is now supported

- Relative imports (`from . import module`) are now supported

- Add support for namespace packages when those are installed using option `--single-version-externally-managed` (part of setuptools/distribute)

1.27 0.8

This is a minor feature release

Features:

- Initial support for Python 3.x
- It is now possible to run the test suite using `python setup.py test`.

  (The actual test suite is still fairly minimal though)

CHAPTER 2

License

2.1 MIT License

CHAPTER 3

Commandline tools

The package can be used as a script using “python -mmodulegraph”.

This script calculates the module graph for the scripts passed on the commandline and by default prints a list of modules in the objectgraph, and their type and location.

The script has a number of options to change the output:

- `-d`: Increase the debug level
- `-q`: Clear the debug level (emit minimal output)
- `-m`: The arguments are module names instead of script files
- `-x name`: Add `name` to the list of excludes
- `-p path`: Add `path` to the module search path
- `-g`: Emit a `.dot` file instead of a list of modules
- `-h`: Emit a `.html` file instead of a list of modules

3.1 Deprecation warning

The package also installs a command-line tool named “modulegraph”, this command-line tool is deprecated and will be removed in a future version.

CHAPTER 4

`modulegraph.modulegraph` — Find modules used by a script

This module defines *`ModuleGraph`*, which is used to find the dependencies of scripts using bytecode analysis.

A number of APIs in this module refer to filesystem path. Those paths can refer to files inside zipfiles (for example when there are zipped egg files on `sys.path`). Filenames referring to entries in a zipfile are not marked any way, if `"somepath.zip"` refers to a zipfile, that is `"somepath.zip/embedded/file"` will be used to refer to `embedded/file` inside the zipfile.

4.1 The actual graph

class `modulegraph.modulegraph.`**`ModuleGraph`**`(`[*path*[, *excludes*[, *replace_paths*[, *implies*[, *graph*[, *debug*]]]]]]`)`

Create a new ModuleGraph object. Use the *`run_script()`* method to add scripts, and their dependencies to the graph.

Parameters

- **`path`** – Python search path to use, defaults to `sys.path`
- **`excludes`** – Iterable with module names that should not be included as a dependency
- **`replace_paths`** – List of pathname rewrites `(old, new)`. When this argument is supplied the `co_filename` attributes of code objects get rewritten before scanning them for dependencies.
- **`implies`** – Implied module dependencies, a mapping from a module name to the list of modules it depends on. Use this to tell modulegraph about dependencies that cannot be found by code inspection (such as imports from C code or using the `__import__()` function).
- **`graph`** – A precreated `Graph` object to use, the default is to create a new one.
- **`debug`** – The `ObjectGraph` debug level.

run_script (*pathname*[, *caller*])
: Create, and return, a node by path (not module name). The *pathname* should refer to a Python source file and will be scanned for dependencies.

  The optional argument *caller* is the the node that calls this script, and is used to add a reference in the graph.

import_hook (*name[[, caller[, fromlist[, level, [, attr]]]]]*)
: Import a module and analyse its dependencies

  Parameters

  - **name** – The module name
  - **caller** – The node that caused the import to happen
  - **fromlist** – The list of names to import, this is an empty list for `import name` and a list of names for `from name import a, b, c`.
  - **level** – The import level. The value should be `-1` for classical Python 2 imports, `0` for absolute imports and a positive number for relative imports (where the value is the number of leading dots in the imported name).
  - **attr** – Attributes for the graph edge.

implyNodeReference (*node*, *other*, *edgeData=None*)
: Explictly mark that *node* depends on *other*. Other is either a *`node`* or the name of a module that will be searched for as if it were an absolute import.

createReference (*fromnode*, *tonode*[, *edge_data*])
: Create a reference from *fromnode* to *tonode*, with optional edge data.

  The default for *edge_data* is `"direct"`.

getReferences (*fromnode*)
: Yield all nodes that *fromnode* refers to. That is, all modules imported by *fromnode*.

  Node `None` is the root of the graph, and refers to all notes that were explicitly imported by *`run_script()`* or *`import_hook()`*, unless you use an explicit parent with those methods.

  New in version 0.11.

getReferers (*tonode*, *collapse_missing_modules=True*)
: Yield all nodes that refer to *tonode*. That is, all modules that import *tonode*.

  If *collapse_missing_modules* is false this includes refererences from *`MissingModule`* nodes, otherwise *`MissingModule`* nodes are replaced by the "real" nodes that reference this missing node.

  New in version 0.12.

foldReferences (*pkgnode*)
: Hide all submodule nodes for package *pkgnode* and add ingoing and outgoing edges to *pkgnode* based on the edges from the submodule nodes.

  This can be used to simplify a module graph: after folding 'email' all references to modules in the 'email' package are references to the package.

findNode (*name*)
: Find a node by identifier. If a node by that identifier exists, it will be returned.

  If a lazy node exists by that identifier with no dependencies (excluded), it will be instantiated and returned.

  If a lazy node exists by that identifier with dependencies, it and its dependencies will be instantiated and scanned for additional depende

create_xref([*out*])
Write an HTML file to the *out* stream (defaulting to `sys.stdout`).

The HTML file contains a textual description of the dependency graph.

graphreport([*fileobj*[, *flatpackages*]])

Todo: To be documented

---

report()
Print a report to stdout, listing the found modules with their paths, as well as modules that are missing, or seem to be missing.

4.1.1 Mostly internal methods

The methods in this section should be considered as methods for subclassing at best, please let us know if you need these methods in your code as they are on track to be made private methods before the 1.0 release.

Warning: The methods in this section will be refactored in a future release, the current architecture makes it unnecessarily hard to write proper tests.

class modulegraph.modulegraph.**ModuleGraph**

_determine_parent(*caller*)
Returns the node of the package root voor *caller*. If *caller* is a package this is the node itself, if the node is a module in a package this is the node of for the package and otherwise the *caller* is not a package and the result is `None`.

_find_head_package(*parent*, *name*[, *level*])

Todo: To be documented

_load_tail(*mod*, *tail*)
This method is called to load the rest of a dotted name after loading the root of a package. This will import all intermediate modules as well (using `import_module()`), and returns the module *`node`* for the requested node.

Note: When *tail* is empty this will just return *mod*.

Parameters

- **mod** – A start module (instance of *`Node`*)
- **tail** – The rest of a dotted name, can be empty

Raises **`ImportError`** – When the requested (or one of its parents) module cannot be found

Returns the requested module

_ensure_fromlist (*m*, *fromlist*)
Yield all submodules that would be imported when importing *fromlist* from *m* (using `from m import fromlist...`).

m must be a package and not a regular module.

_find_all_submodules (*m*)
Yield the filenames for submodules of in the same package as *m*.

_import_module (*partname*, *fqname*, *parent*)
Perform import of the module with basename *partname* (`path`) and full name *fqname* (`os.path`). Import is performed by *parent*.

This will create a reference from the parent node to the module node and will load the module node when it is not already loaded.

_load_module (*fqname*, *fp*, *pathname*, *(suffix*, *mode*, *type)*)
Load the module named *fqname* from the given *pathame*. The argument *fp* is either `None`, or a stream where the code for the Python module can be loaded (either byte-code or the source code). The *(suffix, mode, type)* tuple are the suffix of the source file, the open mode for the file and the type of module.

Creates a node of the right class and processes the dependencies of the *`node`* by scanning the byte-code for the node.

Returns the resulting *`node`*.

_scan_code (*code*, *m*)
Scan the *code* object for module *m* and update the dependencies of *m* using the import statemets found in the code.

This will automaticly scan the code for nested functions, generator expressions and list comprehensions as well.

_load_package (*fqname*, *pathname*)
Load a package directory.

_find_module (*name*, *path*[, *parent*])
Locates a module named *name* that is not yet part of the graph. This method will raise `ImportError` when the module cannot be found or when it is already part of the graph. The *name* can not be a dotted name.

The *path* is the search path used, or `None` to use the default path.

When the *parent* is specified *name* refers to a subpackage of *parent*, and *path* should be the search path of the parent.

Returns the result of the global function *`find_module`*.

itergraphreport ([*name*[, *flatpackages*]])

Todo: To be documented

_replace_paths_in_code (*co*)
Replace the filenames in code object *co* using the *replace_paths* value that was passed to the contructor. Returns the rewritten code object.

_calc_setuptools_nspackages ()
Returns a mapping from package name to a list of paths where that package can be found in `--single-version-externally-managed` form.

This method is used to be able to find those packages: these use a magic `.pth` file to ensure that the package is added to `sys.path`, as they do not contain an `__init__.py` file.

Packages in this form are used by system packages and the “pip” installer.

4.2 Graph nodes

The *`ModuleGraph`* contains nodes that represent the various types of modules.

class `modulegraph.modulegraph.`**`Alias`**(*value*)

This is a subclass of string that is used to mark module aliases.

class `modulegraph.modulegraph.`**`Node`**(*identifier*)

Base class for nodes, which provides the common functionality.

Nodes can by used as mappings for storing arbitrary data in the node.

Nodes are compared by comparing their *identifier*.

`debug`

Debug level (integer)

`graphident`

The node identifier, this is the value of the *identifier* argument to the constructor.

`identifier`

The node identifier, this is the value of the *identifier* argument to the constructor.

`filename`

The filename associated with this node.

`packagepath`

The value of `__path__` for this node.

`code`

The `code object` associated with this node

`globalnames`

The set of global names that are assigned to in this module. This includes those names imported through startimports of Python modules.

`startimports`

The set of startimports this module did that could not be resolved, ie. a startimport from a non-Python module.

`__contains__`(*name*)

Return if there is a value associated with *name*.

This method is usually accessed as `name in aNode`.

`__setitem__`(*name*, *value*)

Set the value of *name* to *value*.

This method is usually accessed as `aNode[name] = value`.

`__getitem__`(*name*)

Returns the value of *name*, raises `KeyError` when it cannot be found.

This method is usually accessed as `value = aNode[name]`.

get (*name*[, *default*])
Returns the value of *name*, or the default value when it cannot be found. The *default* is `None` when not specified.

infoTuple ()
Returns a tuple with information used in the `repr()` output for the node. Subclasses can add additional informations to the result.

class `modulegraph.modulegraph.`**AliasNode** (*name*, *node*)
A node that represents an alias from a name to another node.

The value of attribute *graphident* for this node will be the value of *name*, the other *`Node`* attributed are references to those attributed in *node*.

class `modulegraph.modulegraph.`**BadModule** (*identifier*)
Base class for nodes that should be ignored for some reason

class `modulegraph.modulegraph.`**ExcludedModule** (*identifier*)
A module that is explicitly excluded.

class `modulegraph.modulegraph.`**MissingModule** (*identifier*)
A module that is imported but cannot be located.

class `modulegraph.modulegraph.`**InvalidRelativeImport** (*relative_path*, *from_name*)
A module that was imported using a relative import statement that references a file outside of a toplevel package.

class `modulegraph.modulegraph.`**Script** (*filename*)
A python script.

filename
The filename for the script

class `modulegraph.modulegraph.`**BaseModule** (*name*[, *filename*[, *path*]])
The base class for actual modules. The *name* is the possibly dotted module name, *filename* is the filesystem path to the module and *path* is the value of `__path__` for the module.

graphident
The name of the module

filename
The filesystem path to the module.

path
The value of `__path__` for this module.

class `modulegraph.modulegraph.`**BuiltinModule** (*name*)
A built-in module (one in `sys.builtin_module_names`).

class `modulegraph.modulegraph.`**SourceModule** (*name*)
A module for which the python source code is available.

class `modulegraph.modulegraph.`**InvalidSourceModule** (*name*)
A module for which the python source code is available, but where that source code cannot be compiled (due to syntax errors).

This is a subclass of *`SourceModule`*.

New in version 0.12.

class `modulegraph.modulegraph.`**CompiledModule** (*name*)
A module for which only byte-code is available.

class `modulegraph.modulegraph.`**Package** (*name*)
Represents a python package

class `modulegraph.modulegraph.`**`NamespacePackage`**(*name*)

Represents a python namespace package.

This is a subclass of *`Package`*.

class `modulegraph.modulegraph.`**`Extension`**(*name*)

A native extension

Warning: A number of other node types are defined in the module. Those modules aren't used by modulegraph and will be removed in a future version.

4.3 Edge data

The edges in a module graph by default contain information about the edge, represented by an instance of *`DependencyInfo`*.

class `modulegraph.modulegraph.`**`DependencyInfo`**(*conditional*, *function*, *tryexcept*, *fromlist*)

This class is a `namedtuple` for representing the information on a dependency between two modules.

All attributes can be used to deduce if a dependency is essential or not, and are particularly useful when reporting on missing modules (dependencies on *`MissingModule`*).

`fromlist`

A boolean that is true iff the target of the edge is named in the "import" list of a "from" import ("from package import module").

When the target module is imported multiple times this attribute is false unless all imports are in "import" list of a "from" import.

`function`

A boolean that is true iff the import is done inside a function definition, and is false for imports in module scope (or class scope for classes that aren't definined in a function).

`tryexcept`

A boolean that is true iff the import that is done in the "try" or "except" block of a try statement (but not in the "else" block).

`conditional`

A boolean that is true iff the import is done in either block of an "if" statement.

When the target of the edge is imported multiple times the *`function`*, *`tryexcept`* and *`conditional`* attributes of all imports are merged: when there is an import where all these attributes are false the attributes are false, otherwise each attribute is set to true if it is true for at least one of the imports.

For example, when a module is imported both in a try-except statement and furthermore is imported in a function (in two separate statements), both *`tryexcept`* and *`function`* will be true. But if there is a third unconditional toplevel import for that module as well all three attributes are false.

Warning: All attributes but *`fromlist`* will be false when the source of a dependency is scanned from a byte-compiled module instead of a python source file. The *`fromlist`* attribute will stil be set correctly.

4.4 Utility functions

`modulegraph.modulegraph.`**`find_module`**(*name*[, *path*])

A version of `imp.find_module()` that works with zipped packages (and other **PEP 302** importers).

`modulegraph.modulegraph.`**`moduleInfoForPath`**(*path*)

Return the module name, readmode and type for the file at *path*, or None if it doesn't seem to be a valid module (based on its name).

`modulegraph.modulegraph.`**`addPackagePath`**(*packagename*, *path*)

Add *path* to the value of `__path__` for the package named *packagename*.

`modulegraph.modulegraph.`**`replacePackage`**(*oldname*, *newname*)

Rename *oldname* to *newname* when it is found by the module finder. This is used as a workaround for the hack that the `_xmlplus` package uses to inject itself in the `xml` namespace.

CHAPTER 5

modulegraph.find_modules — High-level module dependency finding interface

This module provides a high-level interface to the functionality of the modulegraph package.

modulegraph.find_modules.**find_modules**([*scripts*[, *includes*[, *packages*[, *excludes*[, *path*[, *debug*]]]]]])

High-level interface, takes iterables for: scripts, includes, packages, excludes

And returns a *modulegraph.modulegraph.ModuleGraph* instance, python_files, and extensions

python_files is a list of pure python dependencies as modulegraph.Module objects,

extensions is a list of platform-specific C extension dependencies as modulegraph.Module objects

modulegraph.find_modules.**parse_mf_results**(*mf*)

Return two lists: the first one contains the python files in the graph, the second the C extensions.

Parameters **mf** – a *modulegraph.modulegraph.ModuleGraph* instance

5.1 Lower-level functionality

The functionality in this section is much lower level and should probably not be used. It's mostly documented as a convenience for maintainers.

modulegraph.find_modules.**get_implies**()

Return a mapping of implied dependencies. The key is a, possibly dotted, module name and the value a list of dependencies.

This contains hardcoded list of hard dependencies, for example for C extensions in the standard libary that perform imports in C code, which the generic dependency finder cannot locate.

modulegraph.find_modules.**plat_prepare**(*includes*, *packages*, *excludes*)

Updates the lists of includes, packages and excludes for the current platform. This will add items to these lists based on hardcoded platform information.

`modulegraph.find_modules.`**`find_needed_modules`**`(`[*mf*[, *scripts*[, *includes*[, *packages*[, *warn*]]]]]`)`

Feeds the given `ModuleGraph` with the *scripts*, *includes* and *packages* and returns the resulting graph. This function will create a new graph when *mf* is not specified or `None`.

CHAPTER 6

`modulegraph.util` — Utility functions

`modulegraph.util.`**`imp_find_module`**(*name*, *path=None*)

This function has the same interface as `imp.find_module()`, but also works with dotted names.

`modulegraph.util.`**`imp_walk`**(*name*)

yields the namepart and importer information for every part of a dotted module name, and raises `ImportError` when the *name* cannot be found.

The result elements are tuples with two elements, the first is a module name, the second is the result for `imp.find_module()` for that module (taking into account **PEP 302** importers)

Deprecated since version 0.10.

`modulegraph.util.`**`guess_encoding`**(*fp*)

Returns the encoding of a python source file.

CHAPTER 7

`modulegraph.zipio` — Read-only filesystem access

This module contains a number of functions that mirror functions found in `os` and `os.path`, but have support for data inside zipfiles as well as regular filesystem objects.

The *path* argument of all functions below can refer to an object on the filesystem, but can also refer to an entry inside a zipfile. In the latter case, a prefix of *path* will be the name of zipfile while the rest refers to an object in that zipfile. As an example, when `somepath/mydata.zip` is a zipfile the path `somepath/mydata.zip/somefile.txt` will refer to `somefile.txt` inside the zipfile.

`modulegraph.zipio.`**`open`**`(`*path*[, *mode*]`)`

Open a file, like `the built-in open function`.

The *mode* defaults to `"r"` and must be either `"r"` or `"rb"`.

`modulegraph.zipio.`**`listdir`**`(`*path*`)`

List the contents of a directory, like `os.listdir()`.

`modulegraph.zipio.`**`isfile`**`(`*path*`)`

Returns true if *path* exists and refers to a file.

Raises IOError when *path* doesn't exist at all.

Based on `os.path.isfile()`

`modulegraph.zipio.`**`isdir`**`(`*path*`)`

Returns true if *path* exists and refers to a directory.

Raises IOError when *path* doesn't exist at all.

Based on `os.path.isdir()`

`modulegraph.zipio.`**`islink`**`(`*path*`)`

Returns true if *path* exists and refers to a symbolic link.

Raises IOError when *path* doesn't exist at all.

Based on `os.path.islink()`

`modulegraph.zipio.`**`readlink`**`(`*path*`)`

Returns the contents of a symbolic link, like `os.readlink()`.

`modulegraph.zipio.`**`getmtime`**`(`*path*`)`

Returns the last modifiction time of a file or directory, like `os.path.getmtime()`.

`modulegraph.zipio.`**`getmode`**`(`*path*`)`

Returns the UNIX file mode for a file or directory, like the *st_mode* attribute in the result of `os.stat()`, but excluding the file type.

CHAPTER 8

Online Resources

- Sourcecode repository on GitHub
- The issue tracker

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index