pypackagery Documentation

Release 1.0.4

Marko Ristin

Aug 02, 2019

Contents:

Pypackagery packages a subset of a monorepo and determine the dependent packages.

CHAPTER 1

packagery

Package a subset of a monorepo and determine the dependent packages.

class packagery.**Package**
Represent the package as a dependency graph of the initial set of python files from the code base.

The package includes the initial set, the pypi depedendencies and the files from the code base recursively imported from the initial set.

Variables

- **requirements** – pip requirements, mapped by package name
- **rel_paths** – paths to the python files in the package relative to the root directory of the python code base
- **unresolved_modules** – modules which we couldn't find neither in the codebase nor in requirements nor in built-ins.

to_mapping()
Convert the package recursively to a mapping s.t. it can be converted to JSON and similar formats.

Return type Mapping[str, Any]

class packagery.**Requirement**(*name*, *line*)
Represent a requirement in requirements.txt.

Establishes

- `self.line.endswith("\n")`
- `self.name.strip() == self.name`

to_mapping()
Represent the requirement as a mapping that can be directly converted to JSON and similar formats.

Return type Mapping[str, Any]

class packagery.**UnresolvedModule**(*name*, *importer_rel_path*)
Represent a module which was neither in the code base nor in requirements nor in built-ins.

Variables

- **name** – name of the module
- **importer_rel_path** – path to the file that imported the module

`to_mapping`()

Represent the unresolved module as a mapping that can be directly converted to JSON and similar formats.

Return type `Mapping[str, Any]`

`packagery.`**`collect_dependency_graph`**(*root_dir*, *rel_paths*, *requirements*, *module_to_requirement*)

Collect the dependency graph of the initial set of python files from the code base.

Parameters

- **root_dir** (`Path`) – root directory of the codebase such as `/home/marko/workspace/pqry/production/src/py`
- **rel_paths** (`List[Path]`) – initial set of python files that we want to package. These paths are relative to root_dir.
- **requirements** (`Mapping[str, Requirement]`) – requirements of the whole code base, mapped by package name
- **module_to_requirement** (`Mapping[str, str]`) – module to requirement correspondence of the whole code base

Return type *Package*

Returns resolved depedendency graph including the given initial relative paths,

Requires

- `missing_requirements(module_to_requirement, requirements) == []`
- `all(not rel_pth.is_absolute() for rel_pth in rel_paths)`

Ensures

- `all(req.name in requirements for req in result.requirements.values())`
- `all(pth in result.rel_paths for pth in rel_paths)` (Initial relative paths included)

`packagery.`**`missing_requirements`**(*module_to_requirement*, *requirements*)

List requirements from module_to_requirement missing in the `requirements`.

Parameters

- **module_to_requirement** (`Mapping[str, str]`) – parsed `module_to_requiremnt.tsv`
- **requirements** (`Mapping[str, Requirement]`) – parsed `requirements.txt`

Return type `List[str]`

Returns list of requirement names

Ensures

- `len(result) == len(set(result))`

packagery.**output**(*package*, *out=<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>*, *a_format='verbose'*)

Output the dependency graph in the given format.

If no format is set, the verbose format (i.e. with best readability) is chosen by default.

Parameters

- **package** (*Package*) – dependency graph to be output
- **out** (Textio) – stream where to output the dependency graphas string
- **a_format** (str) – format of the output (e.g., "json")

Return type None

Requires

- a_format is not None a_format in FORMATS

packagery.**parse_module_to_requirement**(*text*, *filename='<unknown>'*)

Parse the correspondence between the modules and the pip packages given as tab-separated values.

Parameters

- **text** (str) – content of module_to_requirement.tsv
- **filename** (str) – where we got the module_to_requirement.tsv from (URL or path)

Return type Mapping[str, str]

Returns name of the module -> name of the requirement

Ensures

- all(not val.endswith("\n") for val in result.values())
- all(not key.endswith("\n") for key in result.keys())

packagery.**parse_requirements**(*text*, *filename='<unknown>'*)

Parse requirements file and return package name -> package requirement as in requirements.txt.

Parameters

- **text** (str) – content of the requirements.txt
- **filename** (str) – where we got the requirements.txt from (URL or path)

Return type Mapping[str, *Requirement*]

Returns name of the requirement (*i.e.* pip package) -> parsed requirement

Ensures

- all(val.name == key for key, val in result.items())

packagery.**resolve_initial_paths**(*initial_paths*)

Resolve the initial paths of the dependency graph by recursively adding *.py files beneath given directories.

Parameters **initial_paths** (List[Path]) – initial paths as absolute paths

Return type List[Path]

Returns list of initial files (*i.e.* no directories)

Requires

- all(pth.is_absolute() for pth in initial_paths)

Ensures

- `all(pth in result for pth in initial_paths if pth.is_file())` (Initial files also in result)
- `len(result) >= len(initial_paths) if initial_paths else result == []`
- `all(pth.is_absolute() for pth in result)`
- `all(pth.is_file() for pth in result)`

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index