mkinit Documentation

Release 1.1.0

Jon Crall

Jan 17, 2024

PACKAGE LAYOUT

A tool to autogenerate explicit top-level imports

Read the docs	https://mkinit.readthedocs.io
Github	https://github.com/Erotemic/mkinit
Pypi	https://pypi.org/project/mkinit

Autogenerates `__init__.py` files statically and dynamically. To use the static version simply pip install and run the `mkinit` command with the directory corresponding to the package.

The main page for this project is: https://github.com/Erotemic/mkinit

CHAPTER

ONE

QUICK START

Install mkinit via `pip install mkinit`. Then run:

```
MOD_INIT_PATH=path/to/repo/module/__init__.py
mkinit "$MOD_INIT_PATH"
```

This will display autogenerated code that exposes all top-level imports. Use the `--diff` option to check differences with existing code, and add the `-w` flag to write the result.

CHAPTER

TWO

MKINIT

2.1 mkinit package

2.1.1 Subpackages

2.1.1.1 mkinit.util package

2.1.1.1.1 Submodules

2.1.1.1.1.1 mkinit.util.orderedset module

This file was autogenerated based on code in ubelt

mkinit.util.orderedset.**is_iterable**(*obj*)

Are we being asked to look up a list of things, instead of a single thing? We check for the *__iter__* attribute so that this can cover types that don't have to be known by this module, such as NumPy arrays.

Strings, however, should be considered as atomic values to look up, not iterables. The same goes for tuples, since they are immutable and therefore valid entries.

We don't need to check for the Python 2 *unicode* type, because it doesn't have an *__iter__* attribute anyway.

class mkinit.util.orderedset.**OrderedSet**(*iterable=None*)

Bases: MutableSet, Sequence

An OrderedSet is a custom MutableSet that remembers its order, so that every entry has an index that can be looked up.

Example

```
>>> OrderedSet([1, 1, 2, 3, 2])
OrderedSet([1, 2, 3])
```

copy()

Return a shallow copy of this object.

Example

```
>>> this = OrderedSet([1, 2, 3])
>>> other = this.copy()
>>> this == other
True
>>> this is other
False
```

add(*key*)

Add *key* as an item to this OrderedSet, then return its index.

If *key* is already in the OrderedSet, return the index it already had.

Example

```
>>> oset = OrderedSet()
>>> oset.append(3)
0
>>> print(oset)
OrderedSet([3])
```

append(*key*)

Add *key* as an item to this OrderedSet, then return its index.

If *key* is already in the OrderedSet, return the index it already had.

Example

```
>>> oset = OrderedSet()
>>> oset.append(3)
0
>>> print(oset)
OrderedSet([3])
```

update(*sequence*)

Update the set with the given iterable sequence, then return the index of the last element inserted.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.update([3, 1, 5, 1, 4])
4
>>> print(oset)
OrderedSet([1, 2, 3, 5, 4])
```

index(*key*)

Get the index of a given entry, raising an IndexError if it's not present.

key can be an iterable of entries that is not a string, in which case this returns a list of indices.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.index(2)
1
```

get_loc(*key*)

Get the index of a given entry, raising an IndexError if it's not present.

key can be an iterable of entries that is not a string, in which case this returns a list of indices.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.index(2)
1
```

get_indexer(*key*)

Get the index of a given entry, raising an IndexError if it's not present.

key can be an iterable of entries that is not a string, in which case this returns a list of indices.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.index(2)
1
```

pop()

Remove and return the last element from the set.

Raises KeyError if the set is empty.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.pop()
3
```

discard(*key*)

Remove an element. Do not raise an exception if absent.

The MutableSet mixin uses this to implement the .remove() method, which *does* raise an error when asked to remove a non-existent item.

Example

```
>>> oset = OrderedSet([1, 2, 3])
>>> oset.discard(2)
>>> print(oset)
OrderedSet([1, 3])
>>> oset.discard(2)
>>> print(oset)
OrderedSet([1, 3])
```

clear()

Remove all items from this OrderedSet.

union(**sets*)

Combines all unique items. Each items order is defined by its first appearance.

Example

```
>>> oset = OrderedSet.union(OrderedSet([3, 1, 4, 1, 5]), [1, 3], [2, 0])
>>> print(oset)
OrderedSet([3, 1, 4, 5, 2, 0])
>>> oset.union([8, 9])
OrderedSet([3, 1, 4, 5, 2, 0, 8, 9])
>>> oset | {10}
OrderedSet([3, 1, 4, 5, 2, 0, 10])
```

intersection(**sets*)

Returns elements in common between all sets. Order is defined only by the first set.

Example

```
>>> oset = OrderedSet.intersection(OrderedSet([0, 1, 2, 3]), [1, 2, 3])
>>> print(oset)
OrderedSet([1, 2, 3])
>>> oset.intersection([2, 4, 5], [1, 2, 3, 4])
OrderedSet([2])
>>> oset.intersection()
OrderedSet([1, 2, 3])
```

difference(**sets*)

Returns all elements that are in this set but not the others.

Example

```
>>> OrderedSet([1, 2, 3]).difference(OrderedSet([2]))
OrderedSet([1, 3])
>>> OrderedSet([1, 2, 3]).difference(OrderedSet([2]), OrderedSet([3]))
OrderedSet([1])
>>> OrderedSet([1, 2, 3]) - OrderedSet([2])
OrderedSet([1, 3])
>>> OrderedSet([1, 2, 3]).difference()
OrderedSet([1, 2, 3])
```

issubset(*other*)

Report whether another set contains this set.

Example

```
>>> OrderedSet([1, 2, 3]).issubset({1, 2})
False
>>> OrderedSet([1, 2, 3]).issubset({1, 2, 3, 4})
True
>>> OrderedSet([1, 2, 3]).issubset({1, 4, 3, 5})
False
```

issuperset(*other*)

Report whether this set contains another set.

Example

```
>>> OrderedSet([1, 2]).issuperset([1, 2, 3])
False
>>> OrderedSet([1, 2, 3, 4]).issuperset({1, 2, 3})
True
>>> OrderedSet([1, 4, 3, 5]).issuperset({1, 2, 3})
False
```

symmetric_difference(*other*)

Return the symmetric difference of two OrderedSets as a new set. That is, the new set will contain all elements that are in exactly one of the sets.

Their order will be preserved, with elements from *self* preceding elements from *other*.

Example

```
>>> this = OrderedSet([1, 4, 3, 5, 7])
>>> other = OrderedSet([9, 7, 1, 3, 2])
>>> this.symmetric_difference(other)
OrderedSet([4, 5, 9, 2])
```

_update_items(*items*)

Replace the 'items' list of this OrderedSet with a new one, updating self.map accordingly.

difference_update(**sets*)

Update this OrderedSet to remove items from one or more other sets.

Example

```
>>> this = OrderedSet([1, 2, 3])
>>> this.difference_update(OrderedSet([2, 4]))
>>> print(this)
OrderedSet([1, 3])
```

```
>>> this = OrderedSet([1, 2, 3, 4, 5])
>>> this.difference_update(OrderedSet([2, 4]), OrderedSet([1, 4, 6]))
>>> print(this)
OrderedSet([3, 5])
```

intersection_update(*other*)

Update this OrderedSet to keep only items in another set, preserving their order in this set.

Example

```
>>> this = OrderedSet([1, 4, 3, 5, 7])
>>> other = OrderedSet([9, 7, 1, 3, 2])
>>> this.intersection_update(other)
>>> print(this)
OrderedSet([1, 3, 7])
```

symmetric_difference_update(*other*)

Update this OrderedSet to remove items from another set, then add items from the other set that were not present in this set.

Example

```
>>> this = OrderedSet([1, 4, 3, 5, 7])
>>> other = OrderedSet([9, 7, 1, 3, 2])
>>> this.symmetric_difference_update(other)
>>> print(this)
OrderedSet([4, 5, 9, 2])
```

_abc_impl = <_abc._abc_data object>

2.1.1.1.1.2 mkinit.util.util_diff module

Port of ubelt utilities + difftext wrapper around difflib

mkinit.util.util_diff.**difftext**(*text1*, *text2*, *context_lines=0*, *ignore_whitespace=False*, *colored=False*)

Uses difflib to return a difference string between two similar texts

Parameters

- **text1** (*str*) – old text
- **text2** (*str*) – new text
- **context_lines** (*int*) – number of lines of unchanged context
- **ignore_whitespace** (*bool*)
- **colored** (*bool*) – if true highlight the diff

Returns

formatted difference text message

Return type

str

References

http://www.java2s.com/Code/Python/Utility/IntelligentdiffbetweentextfilesTimPeters.htm

Example

```
>>> # build test data
>>> text1 = 'one\ntwo\nthree'
>>> text2 = 'one\ntwo\nfive'
>>> # execute function
>>> result = difftext(text1, text2)
>>> # verify results
>>> print(result)
- three
+ five
```

Example

```
>>> # build test data
>>> text1 = 'one\ntwo\nthree\n3.1\n3.14\n3.1415\npi\n3.4\n3.5\n4'
>>> text2 = 'one\ntwo\nfive\n3.1\n3.14\n3.1415\npi\n3.4\n4'
>>> # execute function
>>> context_lines = 1
>>> result = difftext(text1, text2, context_lines, colored=True)
>>> # verify results
>>> print(result)
```

mkinit.util.util_diff.**highlight_code**(*text*, *lexer_name='python'*, ***kwargs*)

Highlights a block of text using ANSI tags based on language syntax.

Parameters

- **text** (*str*) – plain text to highlight
- **lexer_name** (*str*) – name of language. eg: python, docker, c++
- ****kwargs** – passed to pygments.lexers.get_lexer_by_name

Returns

text - highlighted text
If pygments is not installed, the plain text is returned.

Return type
str

Example

```
>>> text = 'import mkinit; print(mkinit)'
>>> new_text = highlight_code(text)
>>> print(new_text)
```

2.1.1.1.1.3 mkinit.util.util_import module

This file was autogenerated based on code in ubelt

mkinit.util.util_import.**_parse_static_node_value**(*node*)

Extract a constant value from a node if possible

mkinit.util.util_import.**_extension_module_tags**()

Returns valid tags an extension module might have

Returns
List[str]

mkinit.util.util_import.**_static_parse**(*varname*, *fpath*)

Statically parse the a constant variable from a python file

Example

```
>>> # xdoctest: +SKIP("ubelt dependency")
>>> dpath = ub.Path.appdir('tests/import/staticparse').ensuredir()
>>> fpath = (dpath / 'foo.py')
>>> fpath.write_text('a = {1: 2}')
>>> assert _static_parse('a', fpath) == {1: 2}
>>> fpath.write_text('a = 2')
>>> assert _static_parse('a', fpath) == 2
>>> fpath.write_text('a = "3"')
>>> assert _static_parse('a', fpath) == "3"
>>> fpath.write_text('a = ["3", 5, 6]')
>>> assert _static_parse('a', fpath) == ["3", 5, 6]
```

(continues on next page)

(continued from previous page)

```
>>> fpath.write_text('a = ("3", 5, 6)')
>>> assert _static_parse('a', fpath) == ("3", 5, 6)
>>> fpath.write_text('b = 10' + chr(10) + 'a = None')
>>> assert _static_parse('a', fpath) is None
>>> import pytest
>>> with pytest.raises(TypeError):
>>>     fpath.write_text('a = list(range(10))')
>>>     assert _static_parse('a', fpath) is None
>>> with pytest.raises(AttributeError):
>>>     fpath.write_text('a = list(range(10))')
>>>     assert _static_parse('c', fpath) is None
```

mkinit.util.util_import.**_platform_pylib_exts**()

Returns .so, .pyd, or .dylib depending on linux, win or mac. On python3 return the previous with and without abi (e.g. .cpython-35m-x86_64-linux-gnu) flags. On python2 returns with and without multiarch.

Returns
tuple

mkinit.util.util_import.**_syspath_modname_to_modpath**(*modname*, *sys_path=None*, *exclude=None*)

syspath version of modname_to_modpath

Parameters

- **modname** (*str*) – name of module to find
- **sys_path** (*None | List[str | PathLike]*) – The paths to search for the module. If unspecified, defaults to `sys.path`.
- **exclude** (*List[str | PathLike] | None*) – If specified prevents these directories from being searched. Defaults to None.

Returns
path to the module.

Return type
str

Note: This is much slower than the pkgutil mechanisms.

There seems to be a change to the editable install mechanism: https://github.com/pypa/setuptools/issues/3548 Trying to find more docs about it.

TODO: add a test where we make an editable install, regular install, standalone install, and check that we always find the right path.

Example

```
>>> print(_syspath_modname_to_modpath('xdoctest.static_analysis'))
...static_analysis.py
>>> print(_syspath_modname_to_modpath('xdoctest'))
...xdoctest
>>> # xdoctest: +REQUIRES(CPython)
>>> print(_syspath_modname_to_modpath('_ctypes'))
..._ctypes...
>>> assert _syspath_modname_to_modpath('xdoctest', sys_path=[]) is None
>>> assert _syspath_modname_to_modpath('xdoctest.static_analysis', sys_path=[]) is␣
↪None
>>> assert _syspath_modname_to_modpath('_ctypes', sys_path=[]) is None
>>> assert _syspath_modname_to_modpath('this', sys_path=[]) is None
```

Example

```
>>> # test what happens when the module is not visible in the path
>>> modname = 'xdoctest.static_analysis'
>>> modpath = _syspath_modname_to_modpath(modname)
>>> exclude = [split_modpath(modpath)[0]]
>>> found = _syspath_modname_to_modpath(modname, exclude=exclude)
>>> if found is not None:
>>>     # Note: the basic form of this test may fail if there are
>>>     # multiple versions of the package installed. Try and fix that.
>>>     other = split_modpath(found)[0]
>>>     assert other not in exclude
>>>     exclude.append(other)
>>>     found = _syspath_modname_to_modpath(modname, exclude=exclude)
>>> if found is not None:
>>>     raise AssertionError(
>>>         'should not have found {}.'.format(found) +
>>>         ' because we excluded: {}.'.format(exclude) +
>>>         ' cwd={} '.format(os.getcwd()) +
>>>         ' sys.path={} '.format(sys.path)
>>>     )
```

mkinit.util.util_import.**modname_to_modpath**(*modname*, *hide_init=True*, *hide_main=False*, *sys_path=None*)

Finds the path to a python module from its name.

Determines the path to a python module without directly import it

Converts the name of a module (__name__) to the path (__file__) where it is located without importing the module. Returns None if the module does not exist.

Parameters

- **modname** (*str*) – The name of a module in `sys_path`.
- **hide_init** (*bool*) – if False, __init__.py will be returned for packages. Defaults to True.
- **hide_main** (*bool*) – if False, and `hide_init` is True, __main__.py will be returned for packages, if it exists. Defautls to False.

- **sys_path** (*None | List[str | PathLike]*) – The paths to search for the module. If unspecified, defaults to `sys.path`.

Returns
modpath - path to the module, or None if it doesn't exist

Return type
str | None

Example

```
>>> modname = 'xdoctest.__main__'
>>> modpath = modname_to_modpath(modname, hide_main=False)
>>> assert modpath.endswith('__main__.py')
>>> modname = 'xdoctest'
>>> modpath = modname_to_modpath(modname, hide_init=False)
>>> assert modpath.endswith('__init__.py')
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = basename(modname_to_modpath('_ctypes'))
>>> assert 'ctypes' in modpath
```

mkinit.util.util_import.**normalize_modpath**(*modpath*, *hide_init=True*, *hide_main=False*)

Normalizes __init__ and __main__ paths.

Parameters

- **modpath** (*str | PathLike*) – path to a module
- **hide_init** (*bool*) – if True, always return package modules as __init__.py files otherwise always return the dpath. Defaults to True.
- **hide_main** (*bool*) – if True, always strip away main files otherwise ignore __main__.py. Defaults to False.

Returns
a normalized path to the module

Return type
str | PathLike

Note: Adds __init__ if reasonable, but only removes __main__ by default

---

Example

```
>>> from xdoctest import static_analysis as module
>>> modpath = module.__file__
>>> assert normalize_modpath(modpath) == modpath.replace('.pyc', '.py')
>>> dpath = dirname(modpath)
>>> res0 = normalize_modpath(dpath, hide_init=0, hide_main=0)
>>> res1 = normalize_modpath(dpath, hide_init=0, hide_main=1)
>>> res2 = normalize_modpath(dpath, hide_init=1, hide_main=0)
>>> res3 = normalize_modpath(dpath, hide_init=1, hide_main=1)
>>> assert res0.endswith('__init__.py')
```

(continues on next page)

(continued from previous page)

```
>>> assert res1.endswith('__init__.py')
>>> assert not res2.endswith('.py')
>>> assert not res3.endswith('.py')
```

mkinit.util.util_import.**modpath_to_modname**(*modpath*, *hide_init=True*, *hide_main=False*, *check=True*, *relativeto=None*)

Determines importable name from file path

Converts the path to a module (__file__) to the importable python name (__name__) without importing the module.

The filename is converted to a module name, and parent directories are recursively included until a directory without an __init__.py file is encountered.

Parameters

- **modpath** (*str*) – module filepath
- **hide_init** (*bool, default=True*) – removes the __init__ suffix
- **hide_main** (*bool, default=False*) – removes the __main__ suffix
- **check** (*bool, default=True*) – if False, does not raise an error if modpath is a dir and does not contain an __init__ file.
- **relativeto** (*str | None, default=None*) – if specified, all checks are ignored and this is considered the path to the root module.

Todo:

- **[ ] Does this need modification to support PEP 420?**
  https://www.python.org/dev/peps/pep-0420/

Returns
modname

Return type
str

Raises
ValueError – if check is True and the path does not exist

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = modpath.replace('.pyc', '.py')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == 'xdoctest.static_analysis'
```

Example

```
>>> import xdoctest
>>> assert modpath_to_modname(xdoctest.__file__.replace('.pyc', '.py')) == 'xdoctest
↪'
>>> assert modpath_to_modname(dirname(xdoctest.__file__.replace('.pyc', '.py'))) ==
↪'xdoctest'
```

Example

```
>>> # xdoctest: +REQUIRES(CPython)
>>> modpath = modname_to_modpath('_ctypes')
>>> modname = modpath_to_modname(modpath)
>>> assert modname == '_ctypes'
```

Example

```
>>> modpath = '/foo/libfoobar.linux-x86_64-3.6.so'
>>> modname = modpath_to_modname(modpath, check=False)
>>> assert modname == 'libfoobar'
```

mkinit.util.util_import.**split_modpath**(*modpath*, *check=True*)

Splits the modpath into the dir that must be in PYTHONPATH for the module to be imported and the modulepath relative to this directory.

Parameters

- **modpath** (*str*) – module filepath
- **check** (*bool*) – if False, does not raise an error if modpath is a directory and does not contain an `__init__.py` file.

Returns
(directory, rel_modpath)

Return type
Tuple[str, str]

Raises
ValueError – if modpath does not exist or is not a package

Example

```
>>> from xdoctest import static_analysis
>>> modpath = static_analysis.__file__.replace('.pyc', '.py')
>>> modpath = abspath(modpath)
>>> dpath, rel_modpath = split_modpath(modpath)
>>> recon = join(dpath, rel_modpath)
>>> assert recon == modpath
>>> assert rel_modpath == join('xdoctest', 'static_analysis.py')
```

2.1.1.1.2 Module contents

2.1.2 Submodules

2.1.2.1 mkinit.__main__ module

mkinit.__main__.**main**()

The mkinit CLI main

2.1.2.2 mkinit._tokenize module

Tokenization help for Python programs.

tokenize(readline) is a generator that breaks a stream of bytes into Python tokens. It decodes the bytes according to PEP-0263 for determining source file encoding.

It accepts a readline-like method which is called repeatedly to get the next line of input (or b"" for EOF). It generates 5-tuples with these members:

> the token type (see token.py) the token (a string) the starting (row, column) indices of the token (a 2-tuple of ints) the ending (row, column) indices of the token (a 2-tuple of ints) the original line (string)

It is designed to match the working of the Python tokenizer exactly, except that it produces COMMENT tokens for comments and gives type OP for all operators. Additionally, all token lists start with an ENCODING token which tells you which encoding was used to decode the bytes stream.

mkinit._tokenize.**ISTERMINAL**(*x*)

mkinit._tokenize.**ISNONTERMINAL**(*x*)

mkinit._tokenize.**ISEOF**(*x*)

mkinit._tokenize.**tokenize**(*readline*)

The tokenize() generator requires one argument, readline, which must be a callable object which provides the same interface as the readline() method of built-in file objects. Each call to the function should return one line of input as bytes. Alternatively, readline can be a callable function terminating with StopIteration:

> readline = open(myfile, 'rb').__next__ # Example of alternate readline

The generator produces 5-tuples with these members: the token type; the token string; a 2-tuple (srow, scol) of ints specifying the row and column where the token begins in the source; a 2-tuple (erow, ecol) of ints specifying the row and column where the token ends in the source; and the line on which the token was found. The line passed is the physical line.

The first token sequence will always be an ENCODING token which tells you which encoding was used to decode the bytes stream.

mkinit._tokenize.**generate_tokens**(*readline*)

Tokenize a source reading Python code as unicode strings.

This has the same API as tokenize(), except that it expects the *readline* callable to return str objects instead of bytes.

mkinit._tokenize.**detect_encoding**(*readline*)

The detect_encoding() function is used to detect the encoding that should be used to decode a Python source file. It requires one argument, readline, in the same way as the tokenize() generator.

It will call readline a maximum of twice, and return the encoding used (as a string) and a list of any lines (left as bytes) it has read in.

It detects the encoding from the presence of a utf-8 bom or an encoding cookie as specified in pep-0263. If both a bom and a cookie are present, but disagree, a SyntaxError will be raised. If the encoding cookie is an invalid charset, raise a SyntaxError. Note that if a utf-8 bom is found, 'utf-8-sig' is returned.

If no encoding is specified, then the default of 'utf-8' will be returned.

mkinit._tokenize.**untokenize**(*iterable*)

Transform tokens back into Python source code. It returns a bytes object, encoded using the ENCODING token, which is the first token sequence output by tokenize.

Each element returned by the iterable must be a token sequence with at least two elements, a token number and token value. If only two tokens are passed, the resulting output is poor.

Round-trip invariant for full input:
Untokenized source will match input source exactly

Round-trip invariant for limited input:
Output bytes will tokenize back to the input t1 = [tok[:2] for tok in tokenize(f.readline)] newcode = untokenize(t1) readline = BytesIO(newcode).readline t2 = [tok[:2] for tok in tokenize(readline)] assert t1 == t2

class mkinit._tokenize.**TokenInfo**(*type*, *string*, *start*, *end*, *line*)

Bases: *TokenInfo*

Create new instance of TokenInfo(type, string, start, end, line)

property exact_type

2.1.2.3 mkinit.dynamic_mkinit module

Dynamically generate the import exec

mkinit.dynamic_mkinit.**dynamic_init**(*modname*, *submodules=None*, *dump=False*, *verbose=False*)

Main entry point for dynamic mkinit.

Dynamically import listed util libraries and their attributes. Create reload_subs function.

Using __import__ like this is typically not considered good style However, it is better than import * and this will generate the good file text that can be used when the module is 'frozen"

Note: Dynamic mkinit is for initial development and prototyping, and even then it is not recommended. For production it is strongly recommended to use static mkinit instead of dynamic mkinit.

Example

```
>>> # The easiest way to use this in your code is to add these lines
>>> # to the module __init__ file
>>> from mkinit import dynamic_init
>>> execstr = dynamic_init('mkinit')
>>> print(execstr)
>>> exec(execstr)  # xdoc: +SKIP
```

mkinit.dynamic_mkinit.**_indent**(*str_*, *indent=' '*)

mkinit.dynamic_mkinit.**_excecute_imports**(*module*, *modname*, *imports*, *verbose=False*)

Module Imports

mkinit.dynamic_mkinit.**_execute_fromimport_star**(*module*, *modname*, *imports*, *check_not_imported=False*, *verbose=False*)

Effectively import * statements

The dynamic_init must happen before any * imports otherwise it wont catch anything.

mkinit.dynamic_mkinit.**_make_initstr**(*modname*, *imports*, *from_imports*, *withheader=True*)

Calls the other string makers

mkinit.dynamic_mkinit.**_make_module_header**()

mkinit.dynamic_mkinit.**_make_imports_str**(*imports*, *rootmodname='.'*)

mkinit.dynamic_mkinit.**_make_fromimport_str**(*from_imports*, *rootmodname='.'*)

mkinit.dynamic_mkinit.**_find_local_submodule_names**(*pkgpath*)

mkinit.dynamic_mkinit.**_autogen_write**(*modpath*, *initstr*)

Todo:

- [ ] : replace with code in mkinit/formatting.py

2.1.2.4 mkinit.formatting module

Contains logic for formatting statically / dynamically extracted information into the final product.

mkinit.formatting.**_ensure_options**(*given_options=None*)

Ensures dict contains all formatting options.

Defaults are:

with_attrs (bool): if True, generate module attribute from imports
(Default: True)

with_mods (bool): if True, generate module imports
(Default: True)

with_all (bool): if True, generate an __all__ variable
(Default: True)

relative (bool): if True, generate relative . imports
(Default: False)

mkinit.formatting.**_insert_autogen_text**(*modpath*, *initstr*, *interface=False*)

Creates new text for *__init__.py* containing the autogenerated code.

If an *__init__.py* already exists in *modpath*, then it tries to intelligently insert the code without clobbering too much. See *_find_insert_points* for details on this process.

mkinit.formatting.**_find_insert_points**(*lines*)

Searches for the points to insert autogenerated text between.

If the *# <AUTOGEN_INIT>* directive exists, then it is preserved and new text is inserted after it. This text clobbers all other text until the *# <AUTOGEN_INIT>* is reached.

If the explicit tags are not specified, mkinit will only clobber text after one of these patterns:

- A line beginning with a (#) comment
- A multiline (triple-quote) comment
- A line beginning with "from __future__"
- A line beginning with "__version__"

If neither explicit tags or implicit patterns exist, all text is clobbered.

Parameters
lines (*List[str]*) – lines of an *__init__.py* file.

Returns
insert points as starting line, ending line, and any required indentation.

Return type
Tuple[int, int, str]

Examples

```
>>> from mkinit.formatting import *  # NOQA
>>> lines = textwrap.dedent(
    '''
    preserved1 = True
    if True:
        # <AUTOGEN_INIT>
        clobbered2 = True
        # </AUTOGEN_INIT>
    preserved3 = True
    ''').strip('\n').split('\n')
>>> start, end, indent = _find_insert_points(lines)
>>> print(repr((start, end, indent)))
(3, 4, '    ')
```

Examples

```
>>> from mkinit.formatting import *  # NOQA
>>> lines = textwrap.dedent(
    '''
    preserved1 = True
    __version__ = '1.0'
    clobbered2 = True
    ''').strip('\n').split('\n')
>>> start, end, indent = _find_insert_points(lines)
>>> print(repr((start, end, indent)))
(2, 3, '')
```

mkinit.formatting.**_indent**(*text*, *indent=' '*)

mkinit.formatting.**_initstr**(*modname*, *imports*, *from_imports*, *explicit={}*, *protected={}*, *private={}*, *options=None*)

Calls the other string makers

CommandLine

```
python -m mkinit.static_autogen _initstr
```

Parameters

- **modname** (*str*) – the name of the module to generate the init str for
- **imports** (*List[str]*) – list of module-level imports
- **from_imports** (*List[Tuple[str, List[str]]]*) – List of submodules and their imported attributes
- **options** (*dict*) – customize output

CommandLine

```
python -m mkinit.formatting _initstr
```

Example

```
>>> modname = 'foo'
>>> imports = ['.bar', '.baz']
>>> from_imports = [('.bar', ['func1', 'func2'])]
>>> initstr = _initstr(modname, imports, from_imports)
>>> print(initstr)
from foo import bar
from foo import baz

from foo.bar import (func1, func2,)

__all__ = ['bar', 'baz', 'func1', 'func2']
```

Example

```
>>> modname = 'foo'
>>> imports = ['.bar', '.baz']
>>> from_imports = [('.bar', list(map(chr, range(97, 123))))]
>>> initstr = _initstr(modname, imports, from_imports)
>>> print(initstr)
from foo import bar
from foo import baz

from foo.bar import (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s,
                     t, u, v, w, x, y, z,)

__all__ = ['a', 'b', 'bar', 'baz', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
           'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
           'y', 'z']
```

Example

```
>>> import pytest
>>> import sys
>>> if sys.version_info < (3, 7):
>>>     pytest.skip('lazy only works on 3.7+')
>>> modname = 'foo'
>>> imports = ['.bar', '.baz']
>>> from_imports = [('.bar', ['func1', 'func2'])]
>>> options = {'lazy_import': 1, 'lazy_boilerplate': None}
>>> initstr = _initstr(modname, imports, from_imports, options=options)
>>> print(initstr)
...
```

```
>>> options = {'lazy_import': 1, 'lazy_boilerplate': 'from importlib import lazy_
↪import'}
>>> initstr = _initstr(modname, imports, from_imports, options=options)
>>> print(initstr.replace('\n\n', '\n'))
from importlib import lazy_import
__getattr__ = lazy_import(
    __name__,
    submodules={
        'bar',
        'baz',
    },
    submod_attrs={
        'bar': [
            'func1',
            'func2',
        ],
    },
)
def __dir__():
    return __all__
__all__ = ['bar', 'baz', 'func1', 'func2']
```

mkinit.formatting.**_make_imports_str**(*imports*, *rootmodname='.'*)

mkinit.formatting.**_packed_rhs_text**(*lhs_text*, *rhs_text*)

packs rhs text to have indentation that agrees with lhs text

Example

```
>>> normname = 'this.is.a.module'
>>> fromlist = ['func{}'.format(d) for d in range(10)]
>>> indent = ''
>>> lhs_text = indent + 'from {normname} import ('.format(
>>>     normname=normname)
>>> rhs_text = ', '.join(fromlist) + ',)'
>>> packstr = _packed_rhs_text(lhs_text, rhs_text)
>>> print(packstr)
```

```
>>> normname = 'this.is.a.very.long.modnamethatwilkeepgoingandgoing'
>>> fromlist = ['func{}'.format(d) for d in range(10)]
>>> indent = ''
>>> lhs_text = indent + 'from {normname} import ('.format(
>>>     normname=normname)
>>> rhs_text = ', '.join(fromlist) + ',)'
>>> packstr = _packed_rhs_text(lhs_text, rhs_text)
>>> print(packstr)
```

```
>>> normname = 'this.is.a.very.long.
↪modnamethatwilkeepgoingandgoingandgoingandgoingandgoingandgoing'
>>> fromlist = ['func{}'.format(d) for d in range(10)]
>>> indent = ''
>>> lhs_text = indent + 'from {normname} import ('.format(
>>>     normname=normname)
>>> rhs_text = ', '.join(fromlist) + ',)'
>>> packstr = _packed_rhs_text(lhs_text, rhs_text)
>>> print(packstr)
```

mkinit.formatting.**_make_fromimport_str**(*from_imports*, *rootmodname='.'*, *indent=''*)

Parameters

- **from_imports** (*list*) – each item is a tuple with module and a list of imported with_attrs.
- **rootmodname** (*str*) – name of root module
- **indent** (*str*) – initial indentation

Example

```
>>> from_imports = [
...     ('.foo', list(map(chr, range(97, 123)))),
...     ('.bar', []),
...     ('.a_longer_package', list(map(chr, range(65, 91)))),
... ]
>>> from_str = _make_fromimport_str(from_imports, indent=' ' * 8)
>>> print(from_str)
from .foo import (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r,
                  s, t, u, v, w, x, y, z,)
from .a_longer_package import (A, B, C, D, E, F, G, H, I, J, K, L, M,
                               N, O, P, Q, R, S, T, U, V, W, X, Y, Z,)
```

2.1.2.5 mkinit.static_analysis module

A paired down version of static_anslysis from xdoctest

mkinit.static_analysis.**_parse_static_node_value**(*node*)

Extract a constant value from a node if possible

mkinit.static_analysis.**parse_static_value**(*key*, *source=None*, *fpath=None*)

Statically parse a constant variable's value from python code.

TODO: This does not belong here. Move this to an external static analysis library.

Parameters

- **key** (*str*) – name of the variable
- **source** (*str | None*) – python text
- **fpath** (*str | None*) – filepath to read if source is not specified

Example

```
>>> key = 'foo'
>>> source = 'foo = 123'
>>> assert parse_static_value(key, source=source) == 123
>>> source = 'foo = "123"'
>>> assert parse_static_value(key, source=source) == '123'
>>> source = 'foo = [1, 2, 3]'
>>> assert parse_static_value(key, source=source) == [1, 2, 3]
>>> source = 'foo = (1, 2, "3")'
>>> assert parse_static_value(key, source=source) == (1, 2, "3")
>>> source = 'foo = {1: 2, 3: 4}'
>>> assert parse_static_value(key, source=source) == {1: 2, 3: 4}
>>> #parse_static_value('bar', source=source)
>>> #parse_static_value('bar', source='foo=1; bar = [1, foo]')
```

mkinit.static_analysis.**package_modpaths**(*pkgpath*, *with_pkg=False*, *with_mod=True*, *followlinks=True*, *recursive=True*, *with_libs=False*, *check=True*)

Finds sub-packages and sub-modules belonging to a package.

Parameters

- **pkgpath** (*str*) – path to a module or package
- **with_pkg** (*bool*) – if True includes package __init__ files (default = False)
- **with_mod** (*bool*) – if True includes module files (default = True)
- **exclude** (*list*) – ignores any module that matches any of these patterns
- **recursive** (*bool*) – if False, then only child modules are included
- **with_libs** (*bool*) – if True then compiled shared libs will be returned as well
- **check** (*bool*) – if False, then then pkgpath is considered a module even if it does not contain an __init__ file.

Yields

str – module names belonging to the package

References

http://stackoverflow.com/questions/1707709/list-modules-in-py-package

Example

```
>>> from mkinit.static_analysis import *
>>> pkgpath = util_import.modname_to_modpath('mkinit')
>>> paths = list(package_modpaths(pkgpath))
>>> print('\n'.join(paths))
>>> names = list(map(util_import.modpath_to_modname, paths))
>>> assert 'mkinit.static_mkinit' in names
>>> assert 'mkinit.__main__' in names
>>> assert 'mkinit' not in names
>>> print('\n'.join(names))
```

mkinit.static_analysis.**is_balanced_statement**(*lines*)

Checks if the lines have balanced parens, brakets, curlies and strings

Parameters
lines (*list*) – list of strings

Returns
False if the statement is not balanced

Return type
bool

Doctest

```
>>> assert is_balanced_statement(['print(foobar)'])
>>> assert is_balanced_statement(['foo = bar']) is True
>>> assert is_balanced_statement(['foo = (']) is False
>>> assert is_balanced_statement(['foo = (', "')('"]) is True
>>> assert is_balanced_statement(
...     ['foo = (', "'''", ")]'''", ')']) is True
>>> #assert is_balanced_statement(['foo = ']) is False
>>> #assert is_balanced_statement(['== ']) is False
```

mkinit.static_analysis.**_locate_ps1_linenos**(*source_lines*)

Determines which lines in the source begin a “logical block” of code.

Note: implementation taken from xdoctest.parser

Parameters
source_lines (*list*) – lines belonging only to the doctest src these will be unindented, prefixed, and without any want.

Returns

a list of indices indicating which lines
are considered “PS1” and a flag indicating if the final line should be considered for a got/want assertion.

Return type
(list, bool)

Example

```
>>> source_lines = ['>>> def foo():', '>>>     return 0', '>>> 3']
>>> linenos, eval_final = _locate_ps1_linenos(source_lines)
>>> assert linenos == [0, 2]
>>> assert eval_final is True
```

Example

```
>>> source_lines = ['>>> x = [1, 2, ', '>>> 3, 4]', '>>> print(len(x))']
>>> linenos, eval_final = _locate_ps1_linenos(source_lines)
>>> assert linenos == [0, 2]
>>> assert eval_final is True
```

mkinit.static_analysis.**_workaround_16806**(*ps1_linenos*, *exec_source_lines*)

workaround for python issue 16806 (https://bugs.python.org/issue16806)

Issue causes lineno for multiline strings to give the line they end on, not the line they start on. A patch for this issue exists *https://github.com/python/cpython/pull/1800*

Note: Starting from the end look at consecutive pairs of indices to inspect the statment it corresponds to. (the first statment goes from ps1_linenos[-1] to the end of the line list.

Implementation taken from xdoctest.parser

2.1.2.6 mkinit.static_mkinit module

Static version of `mkinit.dynamic_autogen`

mkinit.static_mkinit.**autogen_init**(*modpath_or_name*, *submodules=None*, *respect_all=True*, *options=None*, *dry=False*, *diff=False*, *recursive=False*)

Autogenerates imports for a package __init__.py file.

Parameters

- **modpath_or_name** (*PathLike | str*) – path to or name of a package module. The path should reference the dirname not the __init__.py file. If specified by name, must be findable from the PYTHONPATH.
- **submodules** (*List[str] | None, default=None*) – if specified, then only these specific submodules are used in package generation. Otherwise, all non underscore prefixed modules are used.
- **respect_all** (*bool, default=True*) – if False the *__all__* attribute is ignored while parsing.
- **options** (*dict | None*) – formatting options; customizes how output is formatted. See *formatting._ensure_options* for defaults.
- **dry** (*bool, default=False*) – if True, the autogenerated string is not written
- **recursive** (*bool, default=False*) – if True, we will autogenerate init files for all subpackages.

Note: This will partially override the __init__ file. By default everything up to the last comment / __future__ import is preserved, and everything after is overriden. For more fine grained control, you can specify XML-like

<AUTOGEN_INIT> and # </AUTOGEN_INIT> comments around the volitle area. If specified only the area between these tags will be overwritten.

To autogenerate a module on demand, its useful to keep a doctr comment in the __init__ file like this

```
python -m mkinit <your_module>
```

Example

```
>>> init_fpath, new_text = autogen_init('mkinit', submodules=None,
>>>                                     respect_all=True,
>>>                                     dry=True)
>>> assert 'autogen_init' in new_text
```

mkinit.static_mkinit.**static_init**(*modpath_or_name*, *submodules=None*, *respect_all=True*, *options=None*)

Returns the autogenerated initialization string. This can either be executed with *exec* or directly copied into the __init__.py file.

2.1.2.7 mkinit.top_level_ast module

class mkinit.top_level_ast.**TopLevelVisitor**

Bases: NodeVisitor

Parses top-level attribute names

References

For other visit_<classname> values see http://greentreesnakes.readthedocs.io/en/latest/nodes.html

CommandLine

```
python ~/code/mkinit/mkinit/top_level_ast.py TopLevelVisitor:1
```

Example

```
>>> from xdoctest import utils
>>> source = utils.codeblock(
...     '''
...     def foo():
...         def subfunc():
...             pass
...     def bar():
...         pass
...     class Spam(object):
...         def eggs(self):
...             pass
...         @staticmethod
```

(continues on next page)

(continued from previous page)

```
...         def hams():
...             pass
...     ''')
>>> self = TopLevelVisitor.parse(source)
>>> print('attrnames = {!r}'.format(sorted(self.attrnames)))
attrnames = ['Spam', 'bar', 'foo']
```

Example

```
>>> # xdoctest: +REQUIRES(PY3)
>>> from mkinit.top_level_ast import *  # NOQA
>>> from xdoctest import utils
>>> source = utils.codeblock(
...     '''
...     async def asyncfoo():
...         var = 1
...     def bar():
...         pass
...     class Spam(object):
...         def eggs(self):
...             pass
...         @staticmethod
...         def hams():
...             pass
...     ''')
>>> self = TopLevelVisitor.parse(source)
>>> print('attrnames = {!r}'.format(sorted(self.attrnames)))
attrnames = ['Spam', 'asyncfoo', 'bar']
```

Example

```
>>> from xdoctest import utils
>>> source = utils.codeblock(
...     '''
...     a = True
...     if a:
...         b = True
...         c = True
...     else:
...         b = False
...     d = True
...     del d
...     ''')
>>> self = TopLevelVisitor.parse(source)
>>> print('attrnames = {!r}'.format(sorted(self.attrnames)))
attrnames = ['a', 'b']
```

Example

```
>>> from xdoctest import utils
>>> source = utils.codeblock(
...     '''
...     try:
...         d = True
...         e = True
...     except ImportError:
...         raise
...     except Exception:
...         d = False
...         f = False
...     else:
...         f = True
...     ''')
>>> self = TopLevelVisitor.parse(source)
>>> print('attrnames = {!r}'.format(sorted(self.attrnames)))
attrnames = ['d', 'f']
```

_register(*name*)

_unregister(*name*)

classmethod parse(*source*)

visit_FunctionDef(*node*)

visit_AsyncFunctionDef(*node*)

visit_ClassDef(*node*)

visit_Assign(*node*)

visit_If(*node*)

Note: elif clauses don't have a special representation in the AST, but rather appear as extra If nodes within the orelse section of the previous one.

---

visit_Try(*node*)

We only care about checking if (a) a variable is defined in the main body, and (b) that the variable is defined in all except blacks that **don't** immediately re-raise.

visit_TryExcept(*node*)

We only care about checking if (a) a variable is defined in the main body, and (b) that the variable is defined in all except blacks that **don't** immediately re-raise.

visit_Delete(*node*)

2.1.3 Module contents

2.1.3.1 The MkInit Module

A tool to autogenerate explicit top-level imports

Read the docs	https://mkinit.readthedocs.io
Github	https://github.com/Erotemic/mkinit
Pypi	https://pypi.org/project/mkinit

Autogenerates `__init__.py` files statically and dynamically. To use the static version simply pip install and run the `mkinit` command with the directory corresponding to the package.

The main page for this project is: https://github.com/Erotemic/mkinit

2.1.3.1.1 Quick Start

Install mkinit via `pip install mkinit`. Then run:

```
MOD_INIT_PATH=path/to/repo/module/__init__.py
mkinit "$MOD_INIT_PATH"
```

This will display autogenerated code that exposes all top-level imports. Use the `--diff` option to check differences with existing code, and add the `-w` flag to write the result.

`mkinit.`**`autogen_init`**(*modpath_or_name*, *submodules=None*, *respect_all=True*, *options=None*, *dry=False*, *diff=False*, *recursive=False*)

Autogenerates imports for a package __init__.py file.

Parameters

- **modpath_or_name** (*PathLike | str*) – path to or name of a package module. The path should reference the dirname not the __init__.py file. If specified by name, must be findable from the PYTHONPATH.
- **submodules** (*List[str] | None, default=None*) – if specified, then only these specific submodules are used in package generation. Otherwise, all non underscore prefixed modules are used.
- **respect_all** (*bool, default=True*) – if False the *__all__* attribute is ignored while parsing.
- **options** (*dict | None*) – formatting options; customizes how output is formatted. See *formatting._ensure_options* for defaults.
- **dry** (*bool, default=False*) – if True, the autogenerated string is not written
- **recursive** (*bool, default=False*) – if True, we will autogenerate init files for all subpackages.

Note: This will partially override the __init__ file. By default everything up to the last comment / __future__ import is preserved, and everything after is overriden. For more fine grained control, you can specify XML-like `# <AUTOGEN_INIT>` and `# </AUTOGEN_INIT>` comments around the volitle area. If specified only the area between these tags will be overwritten.

To autogenerate a module on demand, its useful to keep a doctr comment in the __init__ file like this

```
python -m mkinit <your_module>
```

Example

```
>>> init_fpath, new_text = autogen_init('mkinit', submodules=None,
>>>                                     respect_all=True,
>>>                                     dry=True)
>>> assert 'autogen_init' in new_text
```

mkinit.**dynamic_init**(*modname*, *submodules=None*, *dump=False*, *verbose=False*)

Main entry point for dynamic mkinit.

Dynamically import listed util libraries and their attributes. Create reload_subs function.

Using __import__ like this is typically not considered good style However, it is better than import * and this will generate the good file text that can be used when the module is 'frozen"

Note: Dynamic mkinit is for initial development and prototyping, and even then it is not recommended. For production it is strongly recommended to use static mkinit instead of dynamic mkinit.

Example

```
>>> # The easiest way to use this in your code is to add these lines
>>> # to the module __init__ file
>>> from mkinit import dynamic_init
>>> execstr = dynamic_init('mkinit')
>>> print(execstr)
>>> exec(execstr)  # xdoc: +SKIP
```

mkinit.**static_init**(*modpath_or_name*, *submodules=None*, *respect_all=True*, *options=None*)

Returns the autogenerated initialization string. This can either be executed with *exec* or directly copied into the __init__.py file.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

Symbols

A

C

D

E

G