flutils Documentation

Release v0.2.0

Finite Loop, LLC

Oct 27, 2018

Library

flutils (flew-tills) is a Python3.6, Python3.7+ library which provides various tools that may be common across multiple Python projects.

The various library sections are loaded using *cherry-picking* to minimize the cost of using flutils.

CHAPTER 1

Decorators

flutils offers the following decorators:

flutils.**cached_property**(*func*)

A property decorator that is only computed once per instance and then replaces itself with an ordinary attribute.

Deleting the attribute resets the property.

Example

Code:

```
from flutils import cached_property

class MyClass:

    def __init__(self):
        self.x = 5

    @cached_property
    def y(self):
        return self.x + 1
```

Usage:

```
>>> obj = MyClass()
>>> obj.y
6
>>> obj.y
6
```

New in version 0.2.0

This decorator is a derivative work of cached_property and is:

Also this decorator is a derivative work of cached_property and is:

CHAPTER 2

Modules

flutils offers the following module utility functions.

flutils.**cherry_pick**(*namespace*)

Replace the calling *cherry-pick-definition package module* with a *cherry-picking module*.

Note: For projects where startup time is critical, this function allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential, the use of this function is heavily discouraged due to error messages created during loading being postponed and thus occurring out of context.

Use this function when there is a need to *cherry-pick* modules that will be loaded (unless already loaded) and executed when an attribute is accessed.

Parameters **namespace** (*dict*) – This should always be set to `globals()`

Example

It is recommended to first build the root package (`__init__.py`) as a normally desired root package. (Make sure that no functions or classes are defined. If needed, define these in a submodule). For example (`mymodule/__init__.py`):

```
"""This is the mymodule docstring."""

from mymodule import mysubmoduleone
import mymodule.mysubmoduletwo as two
from mymodule.mysubmodulethree import afunction
from mymodule.mysubmodulethree import anotherfunction as anotherfunc

MYVAL = 123
```

To use the `cherry_pick` function, the root package module (`__init__.py`) must be converted to a *cherry-pick-definition package module*. This example is the result of converting the root package

(above):

```
"""This is the mymodule docstring."""

from flutils import cherry_pick

MYVAL = 123

__attr_map__ = (
    'mymodule.mysubmoduleone',
    'mymodule.mysubmoduletwo,two',
    'mymodule.mysubmodulethree:afunction',
    'mymodule.mysubmodulethree:anotherfunction,anotherfunc'
)
__additional_attrs__ = dict(
    MYVAL=MYVAL
)

cherry_pick(globals())
```

As you can see, the imports were each converted to a *foreign-name* and placed in the `__attr_map__` `tuple`.

Then, `MYVAL` was put in the `__additional_attrs__` dictionary. Use this dictionary to pass any values to *cherry-picking module*.

And finally the `cherry_pick` function was called with `globals()` as the only argument.

The result is the expected usage of `mymodule`:

```
>> import mymodule
>> mymodule.anotherfunc()
foo bar
```

To test if a cherry-picked module has been loaded, or not:

```
>> import sys
>> sys.modules.get('mymodule.mysubmodulethree')
```

If you get nothing back, it means the cherry-picked module has not been loaded.

Please be aware that there are some cases when all of the cherry-picked modules will be loaded automatically. Using any program that automatically inspects the cherry-picking module will cause the all of the cherry-picked modules to be loaded. Programs such as ipython and pycharm will do this.

Return type `None`

`flutils.`**`lazy_import_module`**(*name*, *package=None*)

Lazy import a python module.

Note: For projects where startup time is critical, this function allows for potentially minimizing the cost of loading a module if it is never used. For projects where startup time is not essential then use of this function is heavily discouraged due to error messages created during loading being postponed and thus occurring out of context.

Parameters

- **name** (*str*) – specifies what module to import in absolute or relative terms (e.g. either `pkg.mod` or `..mod`).
- **package** (*str, optional*) – If `name` is specified in relative terms, then the `package` argument must be set to the name of the package which is to act as the anchor for resolving the package name. Defaults to `None`.

Returns The lazy imported `module`. This module will postpone the execution of it's loader until the module has an attribute accessed.

Return type types.ModuleType

Raises `ImportError` – if the given `name` and `package` can not be loaded.

Examples:

```
>> from flutils import lazy_import_module
>> module = lazy_import_module('mymodule')

Relative import:

>> module = lazy_import_module('.mysubmodule', package='mymodule')
```

CHAPTER 3

Objects

flutils offers the following object utility functions.

flutils.**has_any_attrs**(*obj*, **attrs*)

Check if the given `obj` has **ANY** of the given `*attrs`.

Parameters

- **obj** (*Any*) – The object to check.
- ***attrs** (*str*) – The names of the attributes to check.

Returns `True` if any of the given `*attrs` exist on the given `obj`; otherwise, `False`.

Return type bool

Example

```
>>> from flutils import has_any_attrs
>>> has_any_attrs(dict(),'get','keys','items','values','something')
True
```

flutils.**has_any_callables**(*obj*, **attrs*)

Check if the given `obj` has **ANY** of the given `attrs` and are callable.

Parameters

- **obj** (*Any*) – The object to check.
- ***attrs** (*str*) – The names of the attributes to check.

Returns `True` if ANY of the given `*attrs` exist on the given `obj` and ANY are callable; otherwise, `False`.

Return type bool

Example

```
>>> from flutils import has_any_callables
>>> has_any_callables(dict(),'get','keys','items','values','foo')
True
```

flutils.**has_attrs**(*obj*, **attrs*)

Check if given `obj` has all the given `*attrs`.

Parameters

- **obj** (*Any*) – The object to check.
- ***attrs** (*str*) – The names of the attributes to check.

Returns `True` if all the given `*attrs` exist on the given `obj`; otherwise, `False`.

Return type bool

Example

```
>>> from flutils import has_attrs
>>> has_attrs(dict(),'get','keys','items','values')
True
```

flutils.**has_callables**(*obj*, **attrs*)

Check if given `obj` has all the given `attrs` and are callable.

Parameters

- **obj** (*Any*) – The object to check.
- ***attrs** (*str*) – The names of the attributes to check.

Returns `True` if all the given `*attrs` exist on the given `obj` and all are callable; otherwise, `False`.

Return type bool

Example

```
>>> from flutils import has_callables
>>> has_callables(dict(),'get','keys','items','values')
True
```

flutils.**is_list_like**(*obj*)

Check that given `obj` acts like a list and is iterable.

List-like objects are instances of:

- `UserList`
- `Iterator`
- `KeysView`
- `ValuesView`
- `deque`

- `frozenset`
- `list`
- `set`
- `tuple`

List-like objects are **NOT** instances of:

- `None`
- `bool`
- `bytes`
- `ChainMap`
- `Counter`
- `OrderedDict`
- `UserDict`
- `UserString`
- `defaultdict`
- `Decimal`
- `dict`
- `float`
- `int`
- `str`
- etc…

Parameters **obj** (*Any*) – The object to be checked.

Returns `True` if the given `obj` is list-like; `False` otherwise.

Return type bool

Examples

```
>>> from flutils import is_list_like
>>> is_list_like([1, 2, 3])
True
```

```
>>> is_list_like(reversed([1, 2, 4]))
True
```

```
>>> is_list_like('hello')
False
```

```
>>> is_list_like(sorted('hello'))
True
```

`flutils.`**`is_subclass_of_any`**(*obj*, **classes*)

Check if the given `obj` is a subclass of any of the given `*classes`.

Parameters

- **obj** (*Any*) – The object to check.
- ***classes** (*Any*) – The classes to check against.

Returns `True` if the given `obj` is an instance of ANY given `*classes`; otherwise `False`.

Return type bool

Example

```
>>> from flutils import is_subclass_of_any
>>> from collections import ValuesView, KeysView, UserList
>>> obj = dict(a=1, b=2)
>>> is_subclass_of_any(obj.keys(),ValuesView,KeysView,UserList)
True
```

CHAPTER 4

Paths

flutils offers the following path utility functions.

flutils.**chmod**(*path*, *mode_file=None*, *mode_dir=None*, *include_parent=False*)

Change the mode of a path.

This function processes the given `path` with *`normalize_path`*.

If the given `path` does NOT exist, nothing will be done.

This function will **NOT** change the mode of:

- symlinks (symlink targets that are files or directories will be changed)
- sockets
- fifo
- block devices
- char devices

Parameters

- **path** (*`str` or `bytes` or `pathlib.PosixPath` or `pathlib.WindowsPath`*) – The path of the file or directory to have it's mode changed. This value can be a *glob pattern*.
- **mode_file** (*`int`, `optional`*) – The mode applied to the given `path` that is a file or a symlink target that is a file. Defaults to `0o600`.
- **mode_dir** (*`int`, `optional`*) – The mode applied to the given `path` that is a directory or a symlink target that is a directory. Defaults to `0o700`.
- **include_parent** (*`bool`, `optional`*) – A value of `True` will chmod the parent directory of the given `path` that contains a a *glob pattern*. Defaults to `False`.

Returns Nothing

Return type `None`

Examples

```
>>> from flutils import chmod
>>> chmod('~/tmp/flutils.tests.osutils.txt', 0o660)
```

Supports a *glob pattern*. So to recursively change the mode of a directory just do:

```
>>> chmod('~/tmp/**', mode_file=0o644, mode_dir=0o770)
```

To change the mode of a directory's immediate contents:

```
>>> chmod('~/tmp/*')
```

flutils.**chown**(*path*, *user=None*, *group=None*, *include_parent=False*)

Change ownership of a path.

This function processes the given `path` with *`normalize_path`*.

If the given `path` does NOT exist, nothing will be done.

Parameters

- **path** (*`str` or `bytes` or `pathlib.PosixPath` or `pathlib.WindowsPath`*) – The path of the file or directory that will have it's ownership changed. This value can be a *glob pattern*.
- **user** (*`str` or `int`, `optional`*) – The "login name" used to set the owner of `path`. A value of `'-1'` will leave the owner unchanged. Defaults to the "login name" of the current user.
- **group** (*`str` or `int`, `optional`*) – The group name used to set the group of `path`. A value of `'-1'` will leave the group unchanged. Defaults to the current user's group.
- **include_parent** (*`bool`, `optional`*) – A value of `True` will chown the parent directory of the given `path` that contains a *glob pattern*. Defaults to `False`.

Raises

- `OSError` – If the given `user` does not exist as a "login name" for this operating system.
- `OSError` – If the given `group` does not exist as a "group name" for this operating system.

Returns Nothing

Return type `None`

Examples

```
>>> from flutils import chown
>>> chown('~/tmp/flutils.tests.osutils.txt')
```

Supports a *glob pattern*. So to recursively change the ownership of a directory just do:

```
>>> chown('~/tmp/**')
```

To change ownership of all the directory's immediate contents:

```
>>> chown('~/tmp/*', user='foo', group='bar')
```

flutils.**directory_present**(*path*, *mode=None*, *user=None*, *group=None*)

Ensure the state of the given `path` is present and a directory.

This function processes the given `path` with *normalize_path*.

If the given `path` does **NOT** exist, it will be created as a directory.

If the parent paths of the given `path` do not exist, they will also be created with the `mode`, `user` and `group`.

If the given `path` does exist as a directory, the `mode`, `user`, and :`group` will be applied.

Parameters

- **path** (*str or bytes or pathlib.PosixPath or pathlib.WindowsPath*) – The path of the directory.
- **mode** (*int, optional*) – The mode applied to the `path`. Defaults to `0o700`.
- **user** (*str or int, optional*) – The "login name" used to set the owner of the given `path`. A value of `'-1'` will leave the owner unchanged. Defaults to the "login name" of the current user.
- **group** (*str or int, optional*) – The group name used to set the group of the given `path`. A value of `'-1'` will leave the group unchanged. Defaults to the current user's group.

Raises

- `ValueError` – if the given `path` contains a glob pattern.
- `ValueError` – if the given `path` is not an absolute path.
- `FileExistsError` – if the given `path` exists and is not a directory.
- `FileExistsError` – if a parent of the given `path` exists and is not a directory.

Returns

`PosixPath` or `WindowsPath` depending on the system.

Note: `PurePath` objects are immutable. Therefore, any `PosixPath` or `WindowsPath` objects given to this function will not be the same objects returned.

Return type pathlib.PurePath

flutils.**exists_as**(*path*)

Return a string describing the file type if it exists.

This function processes the given `path` with *normalize_path*.

Parameters **path** (*str or bytes or pathlib.PosixPath or pathlib.WindowsPath*) – The path to check existance.

Returns

one of the following values:

`''` if the given `path` does NOT exist; or, is a broken symbolic link; or, other errors (such as permission errors) are propagated.

`'directory'` if the given `path` points to a directory or is a symbolic link pointing to a directory.

`'file'` if the given `path` points to a regular file or is a symbolic link pointing to a regular file.

`'block device'` if the given `path` points to a block device or is a symbolic link pointing to a block device.

`'char device'` if the given `path` points to a character device or is a symbolic link pointing to a character device.

`'FIFO'` if the given `path` points to a FIFO or is a symbolic link pointing to a FIFO.

`'socket'` if the given `path` points to a Unix socket or is a symbolic link pointing to a Unix socket.

Return type str

Example

```
>>> from flutils import exists_as
>>> exists_as('~/tmp')
'directory'
```

`flutils.`**`find_paths`**(*pattern*)

Find all paths that match the given *glob pattern*.

This function pre-processes the given `pattern` with *`normalize_path`*.

Parameters **`pattern`** (*`str or bytes or pathlib.PosixPath or pathlib.WindowsPath`*) – The path to find; which may contain a *glob pattern*.

Returns A generator that yields `pathlib.PosixPath` or `pathlib.WindowsPath`.

Return type generator

Example

```
>>> from flutils import find_paths
>>> list(find_paths('~/tmp/*'))
[PosixPath('/home/test_user/tmp/file_one'),
PosixPath('/home/test_user/tmp/dir_one')]
```

`flutils.`**`get_os_group`**(*name=None*)

Get an operating system group object.

Parameters **`name`** (*`str or int, optional`*) – The "group name" or `gid`. Defaults to the current users's group.

Raises

- `OSError` – If the given `name` does not exist as a "group name" for this operating system.
- `OSError` – If the given `name` is a `gid` and it does not exist.

Returns A tuple like object.

Return type `struct_group`

Example

```
>>> from flutils import get_os_group
>>> get_os_group('bar')
grp.struct_group(gr_name='bar', gr_passwd='*', gr_gid=2001,
gr_mem=['foo'])
```

`flutils.`**`get_os_user`**(*name=None*)

Return an user object representing an operating system user.

Parameters **`name`** (*`str or int, optional`*) – The "login name" or `uid`. Defaults to the current user's "login name".

Raises

- `OSError` – If the given `name` does not exist as a "login name" for this operating system.
- `OSError` – If the given `name` is an `uid` and it does not exist.

Returns A tuple like object.

Return type `struct_passwd`

Example

```
>>> from flutils import get_os_user
>>> get_os_user('foo')
pwd.struct_passwd(pw_name='foo', pw_passwd='********', pw_uid=1001,
pw_gid=2001, pw_gecos='Foo Bar', pw_dir='/home/foo',
pw_shell='/usr/local/bin/bash')
```

`flutils.`**`normalize_path`**(*path*)

Normalize a given path.

The given `path` will be normalized in the following process.

1. `bytes` will be converted to a `str` using the encoding given by `getfilesystemencoding()`.
2. `PosixPath` and `WindowsPath` will be converted to a `str` using the `as_posix()` method.
3. An initial component of ~ will be replaced by that user's home directory.
4. Any environment variables will be expanded.
5. Non absolute paths will have the current working directory from `os.getcwd()` to change the current working directory before calling this function.

6. Redundant separators and up-level references will be normalized, so that `A//B`, `A/B/`, `A/./B` and `A/foo/../B` all become `A/B`.

Parameters **path** (*str or bytes or pathlib.PosixPath or pathlib.WindowsPath*) – The path to be normalized.

Returns

`PosixPath` or `WindowsPath` depending on the system.

Note: `PurePath` objects are immutable. Therefore, any `PosixPath` or `WindowsPath` objects given to this function will not be the same objects returned.

Return type pathlib.PurePath

Example

```
>>> from flutils import normalize_path
>>> normalize_path('~/tmp/foo/../bar')
PosixPath('/home/test_user/tmp/bar')
```

`flutils.`**`path_absent`**(*path*)

Ensure the given `path` does **NOT** exist.

If the given `path` does exist, it will be deleted.

If the given `path` is a directory, this function will recursively delete all of the directory's contents.

This function processes the given `path` with *normalize_path*.

Parameters **path** (*str or bytes or pathlib.PosixPath or pathlib.WindowsPath*) – The path to remove. This value can also be a *glob pattern*.

Returns Nothing

Return type `None`

CHAPTER 5

Strings

flutils offers the following string utility functions.

flutils.**camel_to_underscore**(*text*)

Convert a camel-cased string to a string containing words separated with underscores.

Parameters **text** (*str*) – The camel-cased string to convert.

Returns An underscore separated string.

Return type str

Example

```
>>> from flutils import camel_to_underscore
>>> camel_to_underscore('FooBar')
'foo_bar'
```

flutils.**underscore_to_camel**(*text*, *lower_first=True*)

Convert a string with words separated by underscores to a camel-cased string.

Parameters

- **text** (*str*) – The camel-cased string to convert.
- **lower_first** (*bool, optional*) – Lowercase the first character. Defaults to `True`

Returns A camel-cased string.

Return type str

Examples

```
>>> from flutils import underscore_to_camel
>>> underscore_to_camel('foo_bar')
'fooBar'
```

```
>>> underscore_to_camel('_one__two___',lower_first=False)
'OneTwo'
```

CHAPTER 6

Validation

flutils offers the following validation functions.

flutils.**validate_identifier**(*identifier*, *allow_underscore=True*)

Validate the given string is a proper identifier.

This validator will also raise an error if the given identifier is a keyword or a builtin identifier.

Parameters

- **identifier** (*str*) – The value to be tested.
- **allow_underscore** (*bool, optional*) – A value of `False` will raise an error when the `identifier` has a value that starts with an underscore _. (Use `False` when validating potential `namedtuple` keys) Default: `True`.

Raises

- `SyntaxError` – If the given identifier is invalid.
- `TypeError` – If the given identifier is not a `str`.

Return type `None`

CHAPTER 7

Install

Because flutils has no dependencies, installing flutils is quite easy.

7.1 Requirements

flutils will only work with Python 3.6, 3.7+

7.2 Install with pipenv

```
>>> pipenv intsall flutils
```

7.3 Install with pip

```
>>> pip install flutils
```

7.4 Install from source

1. Clone the repo:

   ```
   >>> git clone https://gitlab.com/finite-loop/flutils.git flutils
   >>> cd flutils
   ```

2. Use the latest release:

   (a) Find the flutils release `{VERSION}` here (e.g. `v0.2.0`):

   (b) Checkout the release version:

```
>>> git checkout tags/{VERSION}
```

3. Install:

```
>>> ./setup.py install
```

CHAPTER 8

Glossary

cherry-pick is a term used within the context of flutils to describe the process of choosing modules that will be lazy-loaded. Meaning, the module (as set in the *foreign-name*) will be loaded (unless already loaded) and executed when an attribute is accessed.

Cherry-picking differs from *tree shaking* in that it does not remove "dead" code. Instead, code is loaded (unless already loaded) and executed when used. Unused code will not be loaded and executed.

cherry-pick-definition package module is a term used within the context of flutils to describe a Python package module (`__init__.py`) which contains an `__attr_map__` attribute and calls the `cherry_pick` function.

`__attr_map__` must be a `tuple` with each row containing a *foreign-name*

This module may also have an optional `__additional_attrs__` attribute which is a `dictionary` of attribute names and values to be passed to the *cherry-picking module*.

This module should not have any functions or classes defined.

cherry-picking module is a term used within the context of flutils to describe a dynamically generated Python module that will load (unless already loaded) and execute a *cherry-picked* module when an attribute (on the cherry-picking module) is accessed.

foreign-name is a term used within the context of flutils to describe a string that contains the full dotted notation to a module. This is used for *cherry-picking* modules.

This full dotted notation can **not** contain any relative references (e.g `'..othermodule'`, `'.mysubmodule'`). However,the `importlib.util.resolve_name` function can be used to generate the full dotted notation string of a relative referenced module in a *cherry-pick-definition package module*:

```
from importlib.util import resolve_name
from flutils import cherry_pick
__attr_map__ = (
    resolve_name('.mysubmodule', __package__)
```

(continues on next page)

(continued from previous page)

```
)
cherry_pick(globals())
```

The *foreign-name* for the `os.path` module is:

```
'os.path'
```

A *foreign-name* may also reference a *module attribute* by using the full dotted notation to the module, followed with a colon `:` and then the desired *module attribute*.

To reference the `dirname` function:

```
'os.path:dirname'
```

A *foreign-name* can also contain an alias which will become the attribute name on the *cherry-picking module*. This attribute (alias) will be bound to the *cherry-picked* module. Follow the pep-8 naming conventions. when creating the the alias. A foreign-name with an alias is just the foreign-name followed by a comma `,` then the alias:

```
'mymodule.mysubmodule:hello,custom_function'
```

Or:

```
'mymodule.mysubmodule,mymodule'
```

Foreign-names are used in a *cherry-picking module* to manage the loading and executing of modules when calling attributes on the *cherry-picking module*.

glob pattern flutils provides functions for working with filesystem paths. Some of these functions offer the ability to find matching paths using "glob patterns".

Glob patterns are Unix shell-style wildcards (pattern), which are **not** the same as regular expressions. The special characters used in shell-style wildcards are:

Pattern	Meaning
`*`	matches everything
`**`	matches any files and zero or more directories and sub directories
`?`	matches any single character
`[seq]`	matches any character in *seq*
`[!seq]`	matches any character not in *seq*

Warning: Using the `**` pattern in large directory trees may consume an inordinate amount of time.

Examples:

- To find all python files in a directory:

```
>>> from flutils import find_paths
>>> list(find_paths('~/tmp/*.py')
[PosixPath('/home/test_user/tmp/one.py')
PosixPath('/home/test_user/tmp/two.py')]
```

- To find all python files in a directory and any subdirectories:

```
>>> list(find_paths('~/tmp/**/*.py')
[PosixPath('/home/test_user/tmp/one.py')
PosixPath('/home/test_user/tmp/two.py')]
PosixPath('/home/test_user/tmp/zero/__init__.py')]
```

- To find all python files that have a 3 character extension:

```
>>> list(find_paths('~/tmp/*.py?')
```

- To find all .pyc and .pyo files:

```
>>> list(find_paths('~/tmp/*.py[co]')
```

- If you want to match an arbitrary literal string that may have any of the patterns, use `glob.escape`:

```
>>> import glob
>>> base = glob.escape('~/a[special]file%s')
>>> list(find_paths(base % '[0-9].txt'))
```

module attribute is an executable statement or a function/class definition. In other words a module attribute is an attribute on a python module that can reference pretty much anything, such as functions, objects, variables, etc...

tree shaking is a term commonly used within JavaScript context to describe the removal of dead code.

CHAPTER 9

Index

CHAPTER 10

Release Notes

10.1 0.2.0

- Added *cached_property*
- Supports Python3.7

Index