flutils Documentation

Release v0.7

Finite Loop, LLC

Jul 23, 2020

UTILITIES FOR:

flutils (flu-tills) is a collection of commonly used utility functions for Python projects.

CHAPTER

ONE

CODECS

flutils contains additional codecs that, when registered, can be used to convert bytes to strings and strings to bytes.

1.1 b64

The `b64` codec will decode bytes to base64 string characters; and will encode strings containing base64 string characters into bytes. Base64 string characters can span multiple lines and may have whitespace indentation.

New in version 0.4.

1.2 Raw UTF-8 Escape

The `raw_utf8_escape` codec will decode a byte string containing escaped UTF-8 hexadecimal into a string with the proper characters. Strings encoded with the `raw_utf8_escape` codec will be of ascii bytes and have escaped UTF-8 hexadecimal used for non printable characters and each character with an `ord` value above `127`

New in version 0.4.

1.3 Registering Codecs

Using any of the above codecs requires registering them with Python by using the following function:

register_codecs()
Register additional codecs.

New in version 0.4.

Return type None

Examples

```
>>> from flutils.codecs import register_codecs
>>> register_codecs()
>>> 'test©'.encode('raw_utf8_escape')
b'test\\xc2\\xa9'
>>> b'test\\xc2\\xa9'.decode('raw_utf8_escape')
```

(continues on next page)

(continued from previous page)

```
'test©'
>>> 'dGVzdA=='.encode('b64')
b'test'
>>> b'test'.decode('b64')
'dGVzdA=='
```

1.4 General

get_encoding(*name=None*, *default='UTF-8'*)

Validate and return the given encoding codec name.

Parameters

- **name** (*str*) – The name of the encoding to validate. if empty or invalid then the value of the given `default` will be returned.
- **default** (*str, optional*) – If set, this encoding name will be returned if the given `name` is invalid. Defaults to: *SYSTEM_ENCODING*. If set to `None` which will raise a `LookupError` if the given `name` is not valid.

Raises

- **LookupError** – If the given `name` is not a valid encoding codec name and the given `default` is set to `None` or an empty string.
- **LookupError** – If the given `default` is not a valid encoding codec name.

Return type `str`

Returns *str* – The encoding codec name.

Example

```
>>> from flutils.codecs import get_encoding
>>> get_encoding()
'utf-8'
```

SYSTEM_ENCODING: str = 'UTF-8'

The default encoding as indicated by the system.

Type str

CHAPTER

TWO

COMMANDS

flutils offers the following utilities for running commands.

run(*command*, *stdout=None*, *stderr=None*, *columns=80*, *lines=24*, *force_dimensions=False*, *interactive=False*, ***kwargs*)

Run the given command line command and return the command's return code.

When the given `command` is executed, the command's stdout and stderr outputs are captured in a pseudo terminal. The captured outputs are then added to this function's `stdout` and `stderr` IO objects.

This function will capture any ANSI escape codes in the output of the given command. This even includes ANSI colors.

Parameters

- **command** (*str, List[str], Tuple[str]*) – The command to execute.
- **stdout** (`typing.IO`, optional) – An input/output stream that will hold the command's `stdout`. Defaults to: `sys.stdout`; which will output the command's `stdout` to the terminal.
- **stderr** (`typing.IO`, optional) – An input/output stream that will hold the command's `stderr`. Defaults to: `sys.stderr`; which will output the command's `stderr` to the terminal.
- **columns** (*int, optional*) – The number of character columns the pseudo terminal may use. If `force_dimensions` is `False`, this will be the fallback columns value when the the current terminal's column size cannot be found. If `force_dimensions` is `True`, this will be actual character column value. Defaults to: `80`.
- **lines** (*int, optional*) – The number of character lines the pseudo terminal may use. If `force_dimensions` is `False`, this will be the fallback lines value when the the current terminal's line size cannot be found. If `force_dimensions` is `True`, this will be actual character lines value. Defaults to: `24`.
- **force_dimensions** (*bool, optional*) – This controls how the given `columns` and `lines` values are to be used. A value of `False` will use the given `columns` and `lines` as fallback values if the current terminal dimensions cannot be successfully queried. A value of `True` will resize the pseudo terminal using the given `columns` and `lines` values. Defaults to: `False`.
- **interactive** (*bool, optional*) – A value of `True` will interactively run the given `command`. Defaults to: `False`.
- ****kwargs** – Any additional key-word-arguments used with `Popen`. `stdout` and `stderr` will not be used if given in `**default_kwargs`. Defaults to: `{}`.

Return type int

Returns *int* – The return value from running the given command

Raises

- **RuntimeError** – When using interactive=True and the bash executable cannot be located.
- **OSError** – Any errors raised when trying to read the pseudo terminal.

Example

An example using *run* in code:

```
from flutils.cmdutils import run
from io import BytesIO
import sys
import os

home = os.path.expanduser('~')
with BytesIO() as stream:
    return_code = run(
        'ls "%s"' % home,
        stdout=stream,
        stderr=stream
    )
    text = stream.getvalue()
text = text.decode(sys.getdefaultencoding())
if return_code == 0:
    print(text)
else:
    print('Error: %s' % text)
```

---

prep_cmd(*cmd*)

Convert a given command into a tuple for use by subprocess.Popen.

Parameters **cmd** (Sequence) – The command to be converted.

This is for converting a command of type string or bytes to a tuple of strings for use by subprocess.Popen.

Example

```
>>> from flutils.cmdutils import prep_cmd
>>> prep_cmd('ls -Flap')
('ls', '-Flap')
```

Return type Tuple[str, ...]

class CompletedProcess

A NamedTuple that holds a completed process' information.

return_code

The process return code.

Type int

stdout
All lines of the `stdout` from the process.
Type str

stderr
All lines of the `stderr` from the process.
Type str

cmd
The command that the process ran.
Type str

class RunCmd(*raise_error=True*, *output_encoding=None*, ***default_kwargs*)
A simple callable that simplifies many calls to `subprocess.run`.

Parameters

- **raise_error** (*bool, optional*) – A value of `True` will raise a `ChildProcessError` if the process, exits with a non-zero return code. Default: `True`
- **output_encoding** (*str, optional*) – If set, the returned `stdout` and `stderr` will be converted to from bytes to a Python string using this given `encoding`. Defaults to: `None` which will use the value from `locale.getpreferredencoding` or, if not set, the value from `sys.getdefaultencoding` will be used. If the given encoding does NOT exist the default will be used.
- ****default_kwargs** – Any `subprocess.Popen` keyword argument.

default_kwargs
The `default_kwargs` passed into the constructor which may be passed on to `subprocess.run`.
Type `NamedTuple`

output_encoding
The encoding used to decode the process output
Type str

__call__(*cmd*, ***kwargs*)
Run the given command and return the result.
Parameters
- **cmd** (`Sequence`) – The command
- ****kwargs** – Any default_kwargs to pass to `subprocess.run`. These default_kwargs will override any `default_kwargs` set in the constructor.

Raises
- **FileNotFoundError** – If the given `cmd` cannot be found.
- **ChildProcessError** – If `raise_error=True` was set in this class' constructor; and, the process (from running the given `cmd`) returns a non-zero value.
- **ValueError** – If the given `**kwargs` has invalid arguments.

Example

```
>>> from flutils.cmdutils import RunCmd
>>> from subprocess import PIPE
>>> import os
>>> run_command = RunCmd(stdout=PIPE, stderr=PIPE)
>>> result = run_command('ls -flap %s' % os.getcwd())
>>> result.return_code
```

(continues on next page)

(continued from previous page)

```
0
>>> result.stdout
...
>>> result = run_command('ls -flap %s' % os.path.expanduser('~'))
```

Return type *CompletedProcess*

CHAPTER

THREE

DECORATORS

flutils offers the following decorators:

@cached_property

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

Deleting the attribute resets the property.

Note: In Python 3.8 the `functools.cached_property` decorator was added. It is recommended to use the built-in `functools.cached_property`; provided you're using Python >= 3.8. *`cached_property`* remains for use with Python 3.6 and 3.7.

Example

Code:

```
from flutils.decorators 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
```

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

FOUR

MODULES

flutils offers the following module utility functions:

cherry_pick (*namespace*)

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

Use this function when there is a need to *cherry-pick* modules. This means the loading and executing, of a module, will be postponed until an attribute is accessed.

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

Return type `None`

> **Warning:** 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.

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 anotherfuc

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 rewriting the root package (above):

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

from flutils.moduleutils import cherry_pick

MYVAL = 123
```

(continues on next page)

(continued from previous page)

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

cherry_pick(globals())
```

As you can see, the imports were each rewritten 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.

---

lazy_import_module(*name*, *package=None*)

Lazy import a python module.

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`.

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

Return type

`Module`

- The lazy imported module with the execution of it's loader postponed until an attribute accessed.

Warning: 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.

Examples

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

Relative import:

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

CHAPTER

FIVE

NAMED TUPLES

flutils offers the following named-tuple utility functions:

to_namedtuple(*obj*)

Convert particular objects into a namedtuple.

Parameters **obj** (Union[List, Mapping, NamedTuple, SimpleNamespace, Tuple]) – The object to be converted (or have it's contents converted) to a NamedTuple.

If the given type is of `list` or `tuple`, each item will be recursively converted to a `NamedTuple` provided the items can be converted. Items that cannot be converted will still exist in the returned object.

If the given type is of `list` the return value will be a new `list`. This means the items are not changed in the given `obj`.

If the given type is of `Mapping` (`dict`), keys that can be proper identifiers will become attributes on the returned `NamedTuple`. Additionally, the attributes of the returned `NamedTuple` are sorted alphabetically.

If the given type is of `OrderedDict`, the attributes of the returned `NamedTuple` keep the same order as the given `OrderedDict` keys.

If the given type is of `SimpleNamespace`, The attributes are sorted alphabetically in the returned `NamedTuple`.

Any identifier (key or attribute name) that starts with an underscore cannot be used as a `NamedTuple` attribute.

All values are recursively converted. This means a dictionary that contains another dictionary, as one of it's values, will be converted to a `NamedTuple` with the attribute's value also converted to a `NamedTuple`.

Return type

`list`

A list with any of it's values converted to a `NamedTuple`.

`tuple`

A tuple with any of it's values converted to a `NamedTuple`.

`NamedTuple`.

Example

```
>>> from flutils.namedtupleutils import to_namedtuple
>>> dic = {'a': 1, 'b': 2}
>>> to_namedtuple(dic)
NamedTuple(a=1, b=2)
```

CHAPTER

SIX

OBJECTS

flutils offers the following object utility functions:

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.

Return type

`bool`

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

Example

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

---

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.

Return type

`bool`

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

Example

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

---

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.

Return type

`bool`

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

Example

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

---

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.

Return type

`bool`

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

Example

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

---

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 check.

Return type

`bool`

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

Examples

```
>>> from flutils.objutils 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
```

`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.

Return type

bool

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

Example

```
>>> from flutils.objutils 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

SEVEN

PACKAGES

flutils offers the following package utilities:

7.1 Version Numbers

In flutils a version number consists of two or three dot-separated numeric components, with an optional "pre-release" tag on the right-component. The pre-release tag consists of the letter 'a' (alpha) or 'b' (beta) followed by a number. If the numeric components of two version numbers are equal, then one with a pre-release tag will always be deemed earlier (lesser) than one without.

The following are valid version numbers:

```
0.4
0.4.1
0.5a1
0.5b3
0.5
0.9.6
1.0
1.0.4a3
1.0.4b1
1.0.4
```

The following are examples of invalid version numbers:

```
1
2.7.2.2
1.3.a4
1.3pl1
1.3c4
```

The following function is designed to work with version numbers formatted as described above:

bump_version(*version*, *position=2*, *pre_release=None*)

Increase the version number from a version number string.

New in version 0.3

Parameters

- **version** (*str*) – The version number to be bumped.
- **position** (*int, optional*) – The position (starting with zero) of the version number component to be increased. Defaults to: 2

- **pre_release** (*str, Optional*) – A value of `a` or `alpha` will create or increase an alpha version number. A value of `b` or `beta` will create or increase a beta version number.

Raises

- **ValueError** – if the given `version` is an invalid version number.
- **ValueError** – if the given `position` does not exist.
- **ValueError** – if the given `prerelease` is not in: `a, alpha, b, beta`
- **ValueError** – if trying to 'major' part, of a version number, to a pre-release version.

Return type

`str`

- The increased version number.

Examples

```
>>> from flutils.packages import bump_version
>>> bump_version('1.2.2')
'1.2.3'
>>> bump_version('1.2.3', position=1)
'1.3'
>>> bump_version('1.3.4', position=0)
'2.0'
>>> bump_version('1.2.3', prerelease='a')
'1.2.4a0'
>>> bump_version('1.2.4a0', pre_release='a')
'1.2.4a1'
>>> bump_version('1.2.4a1', pre_release='b')
'1.2.4b0'
>>> bump_version('1.2.4a1')
'1.2.4'
>>> bump_version('1.2.4b0')
'1.2.4'
>>> bump_version('2.1.3', position=1, pre_release='a')
'2.2a0'
>>> bump_version('1.2b0', position=2)
'1.2.1'
```

CHAPTER

EIGHT

PATHS

flutils offers the following path utility functions:

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`, `bytes` or `Path`) – 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`.

Return type `None`

Examples

```
>>> from flutils.pathutils 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/*')
```

---

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`, `bytes` or `Path`) – 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.

Return type `None`

Examples

```
>>> from flutils.pathutils 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')
```

---

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, bytes or Path) – 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.

Return type

Path

- PosixPath or WindowsPath depending on the system.

Note: Path objects are immutable. Therefore, any given path of type Path will not be the same object returned.

Example

```
>>> from flutils.pathutils import directory_present
>>> directory_present('~/tmp/test_path')
PosixPath('/Users/len/tmp/test_path')
```

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, bytes or Path) – The path to check for existence.

Return type

str

- '' (empty string): 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.

Example

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

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`, `bytes` or `Path`) – The path to find; which may contain a *glob pattern*.

Return type `Generator`

Yields `pathlib.PosixPath` or `pathlib.WindowsPath`

Example

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

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.

Return type

`struct_group`

- A tuple like object.

Example

```
>>> from flutils.pathutils import get_os_group
>>> get_os_group('bar')
```

(continues on next page)

(continued from previous page)

```
grp.struct_group(gr_name='bar', gr_passwd='*', gr_gid=2001,
gr_mem=['foo'])
```

---

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.

Return type

struct_passwd

- A tuple like object.

Example

```
>>> from flutils.pathutils 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')
```

---

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, bytes or Path) – The path to be normalized.

Return type

Path

- PosixPath or WindowsPath depending on the system.

Note: `Path` objects are immutable. Therefore, any given `path` of type `Path` will not be the same object returned.

Example

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

path_absent (*path*)

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

New in version 0.4.

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`, `bytes` or `Path`) – The path to remove.

Return type `None`

Example

```
>>> from flutils.pathutils import path_absent
>>> path_absent('~/tmp/test_path')
```

CHAPTER

NINE

SETUP

9.1 Custom `setup.py` Commands

flutils offers the ability to quickly add additional `setup.py` custom commands to a Python project.

9.1.1 Requirements

Custom Setup Commands can be used in Python projects that are using setuptools with a `setup.cfg` configuration file.

Setup Commands requires that the `setup.py` and `setup.cfg` files exist in the root directory of a typical Python project structure:

```
my_project/
│
├── docs/
│   ├── doc1.rst
│   └── doc2.rst
│
├── my_project/
│   ├── __init__.py
│   ├── module1.py
│   └── module2.py
│
├── setup.cfg
├── setup.py
└── tests/
    ├── test1.py
    └── test2.py
```

To use Custom Setup Commands, the `setup.cfg` file must have, at least, the following defined:

```
[metadata]
name == my_project
```

9.1.2 Custom Command Definitions

The actual command definitions can exist in `setup.cfg` and/or `setup_commands.cfg`. If the `setup_commands.cfg` file exists then the command definitions from in this file will be used; and, command definitions in `setup.cfg` are ignored.

The general idea is to have `setup_commands.cfg` ignored by version control, allowing developers customize command definitions to their specific needs. While command definitions needed for deployment, testing etc can be kept in version control.

Each individual Custom-Setup-Command definition starts with a section header of:

```
[setup.command.<name-of-custom-setup-command>]
```

Underneath the section header the following options can be used:

- `description = <text>`: A short description about the custom setup command. This is displayed with `setup.py --help-commands`
- `command = <text>`: The command line command or commands to execute when the `setup.py` custom command is called.
- `commands = <text>`: The command line command or commands to execute when the `setup.py` custom command is called. If both `command` and `commands` exist, the value of `command` will be executed first.
- `name = <text>`: This option can override the `<name-of-custom-setup-command>` set in the section header.

The following string interpolation variables can be used on all but the `name` option.

- `{name}`: the name of the project as set in the `[metadata]` section `name` option.
- `{setup_dir}`: The full path to the directory that contains the `setup.py` file.
- `{home}`: The full path to the user's home directory.

9.1.3 Example `setup.cfg`

The following example of `setup.cfg` contains the definitions for the `setup.py lint` and `setup.py lint-all` commands:

```
[metadata]
name = my_project

[setup.command.lint]
description = Lint the {name} project files
command = pylint --rcfile={setup_dir}/.pylintrc {setup_dir}/{name}

[setup.command.lint_all]
name = lint-all
description = Lint the {name} project and test files
commands =
    pylint --rcfile={setup_dir}/.pylintrc {setup_dir}/{name}
    pylint --rcfile={setup_dir}/.pylintrc {setup_dir}/tests
```

9.1.4 Implementation

The final step of using Custom Setup Commands is to prepare the commands and tell setuptools.

add_setup_cfg_commands(*setup_kwargs*, *setup_dir=None*)
Add additional custom `setup.py` commands that are defined in `setup.cfg`.

Parameters

- **setup_kwargs** (*dict*) – A dictionary holding the setuptools.setup keyword arguments. (see example below).
- **setup_dir** (`str` or `Path`, optional) – The root directory of the project. (e.g. the directory that contains the `setup.py` file). Defaults to: `None` which will try to determine the directory using the call stack.

Return type `None`

Example

Use in `setup.py` like the following:

```
#!/usr/bin/env python

import os

from setuptools import setup

from flutils.setuputils import add_setup_cfg_commands

setup_kwargs = {}
setup_dir = os.path.dirname(os.path.realpath(__file__))
add_setup_cfg_commands(setup_kwargs, setup_dir=setup_dir)
setup(**setup_kwargs)
```

CHAPTER

TEN

STRINGS

flutils offers the following string utility functions:

as_escaped_unicode_literal(*text*)

Convert the given `text` into a string of escaped Unicode hexadecimal.

Parameters **text** (`str`) – The string to convert.

Return type

`str`

A string with each character of the given `text` converted into an escaped Python literal.

Example

```
>>> from flutils.strutils import as_escaped_unicode_literal
>>> t = '1. '
>>> as_literal(t)
'\\x31\\x2e\\u2605\\x20\\U0001f6d1'
```

as_escaped_utf8_literal(*text*)

Convert the given `text` into a string of escaped UTF8 hexadecimal.

Parameters **text** (`str`) – The string to convert.

Return type

`str`

A string with each character of the given `text` converted into an escaped UTF8 hexadecimal.

Example

```
>>> from flutils.strutils import as_literal_utf8
>>> t = '1. '
>>> as_escaped_utf8_literal(t)
'\\x31\\x2e\\xe2\\x98\\x85\\x20\\xf0\\x9f\\x9b
\\x91'
```

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.

Return type str

Example

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

convert_escaped_unicode_literal(*text*)

Convert any escaped Unicode literal hexadecimal character(s) to the proper character(s).

This function will convert a string, that may contain escaped Unicode literal hexadecimal characters, into a string with the proper characters.

Parameters **text** (str) – The string that may have escaped Unicode hexadecimal.

Return type

str

A string with each escaped Unicode hexadecimal character converted into the proper character.

The following Unicode literal formats are supported:

```
\x31
\u0031
\U00000031
```

Examples

Basic usage:

```
>>> from flutils.strutils import convert_escaped_unicode_literal
>>> a = '\\x31\\x2e\\u2605\\x20\\U0001f6d1'
>>> convert_escaped_unicode_literal(a)
'1. '
```

This function is intended for cases when the value of an environment variable contains escaped Unicode literal characters that need to be converted to proper characters:

```
$ export TEST='\x31\x2e\u2605\x20\U0001f6d1'
$ python
```

```
>>> import os
>>> from flutils.strutils import convert_escaped_unicode_literal
>>> a = os.getenv('TEST')
>>> a
'\\x31\\x2e\\u2605\\x20\\U0001f6d1'
>>> convert_escaped_unicode_literal(a)
'1. '
```

convert_escaped_utf8_literal(*text*)

Convert any escaped UTF-8 hexadecimal character bytes into the proper string characters(s).

This function will convert a string, that may contain escaped UTF-8 literal hexadecimal bytes, into a string with the proper characters.

Parameters **text** (str) – The string that may have escaped UTF8 hexadecimal.

Raises **UnicodeDecodeError** – if any of the escaped hexadecimal characters are not proper UTF8 bytes.

Return type

str

A string with each escaped UTF8 hexadecimal character converted into the proper character.

Examples

Basic usage:

```
>>> from flutils.strutils import convert_raw_utf8_escape
>>> a = 'test\\xc2\\xa9'
>>> convert_escaped_utf8_literal(a)
'test©'
```

This function is intended for cases when the value of an environment variable contains escaped UTF-8 literal characters (bytes) that need to be converted to proper characters:

```
$ export TEST='test\\xc2\\xa9'
$ python
```

```
>>> import os
>>> from flutils.strutils import convert_raw_utf8_escape
>>> a = os.getenv('TEST')
>>> a
'test\\xc2\\xa9'
>>> convert_escaped_utf8_literal(a)
'test©'
```

---

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.

Return type str

Examples

```
>>> from flutils.strutils import underscore_to_camel
>>> underscore_to_camel('foo_bar')
'fooBar'
>>> underscore_to_camel('_one__two___',lower_first=False)
'OneTwo'
```

CHAPTER

ELEVEN

TEXT

flutils offers the following text functions and objects:

len_without_ansi(*seq*)

Return the character length of the given `Sequence` without counting any ANSI codes.

New in version 0.6

Parameters **seq** (`Sequence`) – A string or a list/tuple of strings.

Return type `int`

Example

```
>>> from flutils.txtutils import len_without_ansi
>>> text = '\x1b[38;5;209mfoobar\x1b[0m'
>>> len_without_ansi(text)
6
```

class AnsiTextWrapper(*width=70*, *initial_indent=''*, *subsequent_indent=''*, *expand_tabs=True*, *replace_whitespace=True*, *fix_sentence_endings=False*, *break_long_words=True*, *drop_whitespace=True*, *break_on_hyphens=True*, *tabsize=8*, ***, *max_lines=None*, *placeholder=' [...]'*)

A `TextWrapper` object that correctly wraps text containing ANSI codes.

New in version 0.6

Parameters

- **width** (*int, optional*) – The maximum length of wrapped lines. As long as there are no individual words in the input text longer than this given `width`, *`AnsiTextWrapper`* guarantees that no output line will be longer than `width` characters. Defaults to: `70`
- **initial_indent** (*str, optional*) – Text that will be prepended to the first line of wrapped output. Counts towards the length of the first line. An empty string value will not indent the first line. Defaults to: `''` an empty string.
- **subsequent_indent** (*str, optional*) – Text that will be prepended to all lines of wrapped output except the first. Counts towards the length of each line except the first. Defaults to: `''` an empty string.
- **expand_tabs** (*bool, optional*) – If `True`, then all tab characters in text will be expanded to spaces using the `expandtabs`. Also see the `tabsize` argument. Defaults to: `True`.

- **replace_whitespace** (*bool, optional*) – If `True`, after tab expansion but before wrapping, the wrap() method will replace each whitespace character with a single space. The whitespace characters replaced are as follows: tab, newline, vertical tab, form-feed, and carriage return (`'\t\n\v\f\r'`). Defaults to: `True`.
- **fix_sentence_endings** (*bool, optional*) – If `True`, *AnsiTextWrapper* attempts to detect sentence endings and ensure that sentences are always separated by exactly two spaces. This is generally desired for text in a monospaced font. However, the sentence detection algorithm is imperfect; it assumes that a sentence ending consists of a lowercase letter followed by one of '.', '!', or '?', possibly followed by one of '"' or '"', followed by a space. Defaults to: `False`.
- **break_long_words** (*bool, optional*) – If `True`, then words longer than width will be broken in order to ensure that no lines are longer than width. If it is `False`, long words will not be broken, and some lines may be longer than width. (Long words will be put on a line by themselves, in order to minimize the amount by which width is exceeded.) Defaults to: `True`.
- **drop_whitespace** (*bool, optional*) – If `True`, whitespace at the beginning and ending of every line (after wrapping but before indenting) is dropped. Whitespace at the beginning of the paragraph, however, is not dropped if non-whitespace follows it. If whitespace being dropped takes up an entire line, the whole line is dropped. Defaults to: `True`
- **break_on_hyphens** (*bool, optional*) – If `True`, wrapping will occur preferably on whitespaces and right after hyphens in compound words, as it is customary in English. If `false`, only whitespaces will be considered as potentially good places for line breaks, but you need to set `break_long_words` to `False` if you want truly insecable words. Defaults to: `True`.
- **tabsize** (*int, optional*) – If `expand_tabs` is `True`, then all tab characters in text will be expanded to zero or more spaces, depending on the current column and the given tab size. Defaults to: `8`.
- **max_lines** (`int` or `None`, optional) – If not `None`, then the output will contain at most `max_lines lines`, with `placeholder` appearing at the end of the output. Defaults to: `None`.
- **placeholder** (*str, optional*) – Text that will appear at the end of the output text if it has been truncated. Defaults to: `' [...]'`

Note: The `initial_indent`, `subsequent_indent` and `placeholder` parameters can also contain ANSI codes.

Note: If `expand_tabs` is `False` and `replace_whitespace` is `True`, each tab character will be replaced by a single space, which is not the same as tab expansion.

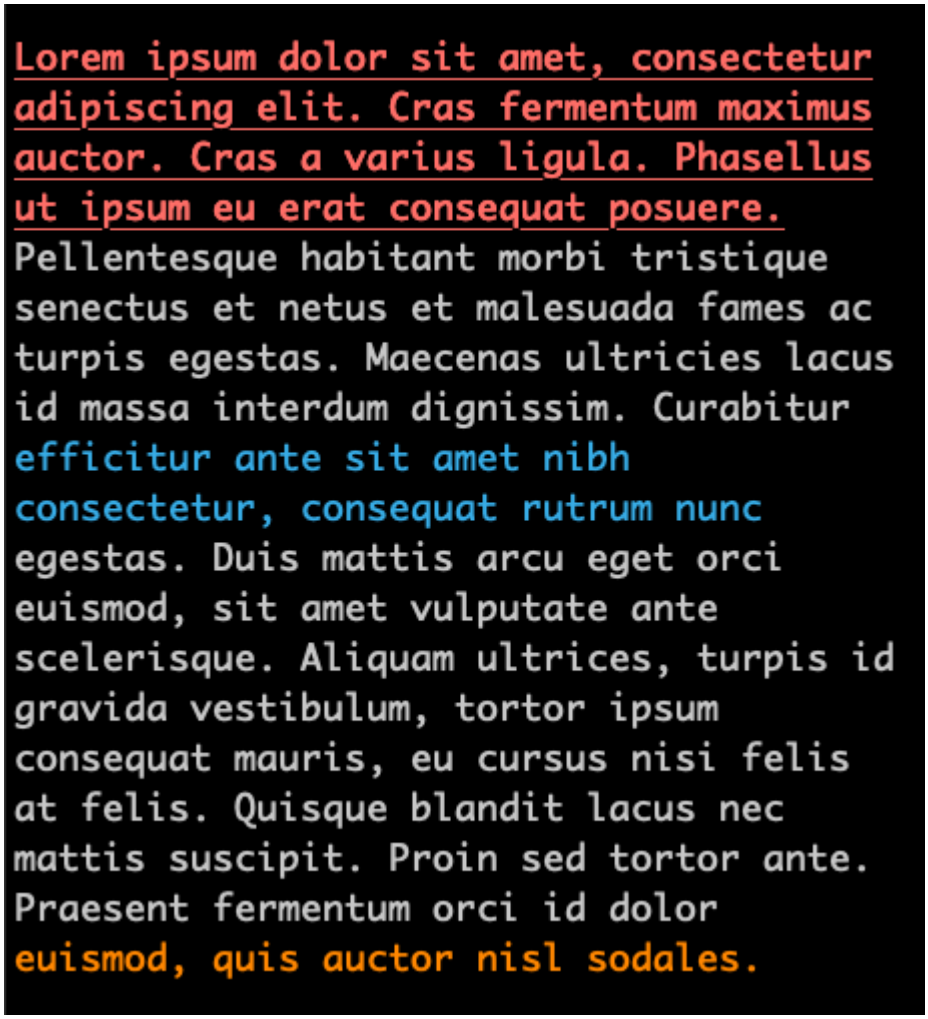
Note: If `replace_whitespace` is `False`, newlines may appear in the middle of a line and cause strange output. For this reason, text should be split into paragraphs (using `str.splitlines` or similar) which are wrapped separately.

Example

Use *AnsiTextWrapper* the same way as using TextWrapper:

```
from flutils.txtutils import AnsiTextWrapper
text = (
    '\x1b[31m\x1b[1m\x1b[4mLorem ipsum dolor sit amet, '
    'consectetur adipiscing elit. Cras fermentum maximus '
    'auctor. Cras a varius ligula. Phasellus ut ipsum eu '
    'erat consequat posuere.\x1b[0m Pellentesque habitant '
    'morbi tristique senectus et netus et malesuada fames ac '
    'turpis egestas. Maecenas ultricies lacus id massa '
    'interdum dignissim. Curabitur \x1b[38;2;55;172;230m '
    'efficitur ante sit amet nibh consectetur, consequat '
    'rutrum nunc\x1b[0m egestas. Duis mattis arcu eget orci '
    'euismod, sit amet vulputate ante scelerisque. Aliquam '
    'ultrices, turpis id gravida vestibulum, tortor ipsum '
    'consequat mauris, eu cursus nisi felis at felis. '
    'Quisque blandit lacus nec mattis suscipit. Proin sed '
    'tortor ante.  Praesent fermentum orci id dolor '
    '\x1b[38;5;208meuismod, quis auctor nisl sodales.\x1b[0m'
)
wrapper = AnsiTextWrapper(width=40)
wrapped_text = wrapper.fill(text)
print(wrapped_text)
```

The output:



---

fill(*text*)

Wraps a single paragraph.

Parameters **text** (str) – The text to be wrapped.

Return type str

wrap(*text*)

Wraps the single paragraph in the given `text` so every line is at most `width` characters long. All wrapping options are taken from instance attributes of the *AnsiTextWrapper* instance.

Parameters **text** (*str*) – The text to be wrapped.

Return type List[str]

Returns A `List[str]` of output lines, without final newlines. If the wrapped output has no content, the returned list is empty.

CHAPTER

TWELVE

VALIDATION

flutils offers the following validation functions.

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 or UserString) – 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) Defaults to: True.

Raises

- **SyntaxError** – If the given identifier is invalid.
- **TypeError** – If the given identifier is not a str or UserString.

Return type None

Example

```
>>> from flutils.validators import validate_identifier
>>> validate_identifier('123')
SyntaxError: The given 'identifier', '123', cannot start with a number
```

CHAPTER

THIRTEEN

INSTALL

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

13.1 Requirements

flutils will only work with Python 3.6, 3.7 and 3.8+

13.2 Install with pip

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

13.3 Install from source

1. Clone the repo:

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

2. Use the latest release:

   1. `{VERSION}` = 'v0.7'
   2. Checkout the release version:

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

3. Install:

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

CHAPTER

FOURTEEN

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__)
)
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

FIFTEEN

INDEX

CHAPTER

SIXTEEN

RELEASE NOTES

16.1 0.7

- Released: 2020-07-23
- Added *get_encoding*
- Added *prep_cmd*
- Added *RunCmd*
- Added *CompletedProcess*

16.2 0.6.1

- Released: 2020-07-11
- Bug fixes:
  - Removed `tests` from builds
- added `py.typed` file for PEP 561

16.3 0.6

- Released: 2020-03-30
- Added *len_without_ansi*
- Added *AnsiTextWrapper*
- Added *run*

16.4 0.5

- Released: 2020-02-20
- Added *to_namedtuple*
- Works with Python3.8

16.5 0.4

- Released: 2019-07-16
- Added *add_setup_cfg_commands*
- Added *as_escaped_unicode_literal*
- Added *as_escaped_utf8_literal*
- Added *convert_escaped_unicode_literal*
- Added *convert_escaped_utf8_literal*
- Added the *b64* codecs via *register_codecs*
- Added the *Raw UTF-8 Escape* codec via *register_codecs*
- Removed the restriction of Python string method names as identifiers in:
  - *validate_identifier*
  - *cherry_pick*
- Rewrite of *bump_version*
- Improved unit and integration tests.

16.6 0.3

- Released: 2018-10-29
- Added *bump_version*

16.7 0.2

- Released: 2018-10-27
- Added *cached_property*
- Works with Python3.7

16.8 0.1

- Released: 2018-07-06
- Initial release
- Works with Python3.6

CHAPTER

SEVENTEEN

DEVELOPMENT

Contributions to flutils can be made by sending a Merge request via fork of GitLab. If you don't have a GitLab account you can sign up here.

17.1 Requirements

17.1.1 pyenv

pyenv is used to install and manage different versions of Python on your system and will not effect the system's Python.

1. Install the pyenv prerequisites.
2. (Optional) By default pyenv will be installed at `~/.pyenv`. If you desire, change the install location by setting the environment variable, `PYENV_ROOT`, with the new location (e.g. `export PYENV_ROOT=/a/new/path`).
3. Run the following to install:

```
>>> curl -L https://github.com/pyenv/pyenv-installer/raw/master/bin/pyenv-
↪installer | bash
```

   - Make sure to follow any post install instructions.
   - Mac users can install pyenv via homebrew. THIS IS NOT RECOMMENDED; because, updates to pyenv lag behind. Use at your own peril.

4. Restart your shell so that path changes take effect:

```
>>> exec "${SHELL}"
```

5. Add the `xxenv-latest` plug-in to `pyenv`:

```
>>> git clone https://github.com/momo-lab/xxenv-latest.git "$(pyenv root)"/
↪plugins/xxenv-latest
```

17.1.2 Python

flutils is designed to work with multiple versions of Python. So, we need to make sure the latest versions are installed.

- As new versions of Python are released you'll need to follow these same instructions.

1. Update pyenv:

```
>>> pyenv update
```

2. Install the latest versions of the following Pythons:

```
>>> pyenv latest install -v
>>> pyenv latest install -v 3.7
>>> pyenv latest install -v 3.6
```

17.1.3 pipenv

pipenv is used for setting up a development virtualenv and installing and managing the development dependencies.

Follow the instructions for installing pipenv here.

Warning: There has not been a new release of pipenv since 2018-11-26. There are also rumors that the project may be dead. Because of this, sometime in the future, flutils will replace the use of pipenv with poetry.

17.2 Setup

17.2.1 Code

1. Clone the flutils code from GitLab :
   - Clone with SSH:

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

   - or, with HTTPS:

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

2. Change directory:

```
>>> cd flutils
```

17.2.2 Virtual Environment

1. In the code's root directory run the following command to setup the virtualenv needed for development:

```
>>> pipenv install --dev --python "$(pyenv root)/versions/$(pyenv latest -p)/bin/
↪python"
```

2. To activate the flutils virtualenv:

```
>>> pipenv shell
```

17.3 Testing

Within the activated flutils virtualenv, the following commands can be used to test code changes:

- `./setup.py test` will run all unit tests, integration tests and type checks (mypy).
- `./setup.py coverage` will run all the tests and produce a coverage report.
- `./setup.py lint` will run code analysis checks using pylint.
- `./setup.py style` will run code styling enforcement checks using flake8.
- `./setup.py security` will run code security checks using bandit.
- `./setup.py pipelinetests` will run all of the above tests.
- `make tests` will run all of the above tests across the multiple supported versions of Python using tox. make sure to run this command before submitting a pull-request. Code that does not pass this test will not be accepted.

17.4 CI Environment

Warning: This requires Docker to be installed and running.

Run the following commands when a new version of Python has been released:

1. If a new minor version (e.g. 3.9) of Python has been released, changes need to be made in `tests/Dockerfile` to reflect the new version.
2. To build a new Docker image with the latest supported versions of Python:

```
>>> make docker-image
```

3. Deploy the newly built Docker image to the regestry:

```
>>> make docker-image-push
```

17.5 New Release

1. Bump the version number in `flutils/__init__.py`, commit and push.
2. Update the build requirements for the documentation build server.

```
>>> make docs-requirements
```

3. Cut a new tag:

```
>>> git tag "v$(python -c 'import flutils; print(flutils.__version__)')"
```

4. Push the new tag:

```
>>> git push --tags
```

5. Build and push the release to `test.pypi.org`:

```
>>> make sdist-push-test
```

6. Go to the link shown in the output of the above command and verify everything.
7. Build and push the release to `pypi.org`:

```
>>> make sdist-push
```

17.6 References

- pyenv installer

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W