# filesystem$_{t}ree.pyDocumentation$

***Release 1.1.2***

**Gratipay, LLC**

**Sep 27, 2017**

# Contents

This is a library for managing a filesystem tree. Test fixture is the driving use-case.

# CHAPTER 1

## Installation, Testing, and License

`filesystem_tree` is available on GitHub and on PyPI:

```
$ pip install filesystem_tree
```

We test against 64-bit Python 2.6, 2.7, 3.2, 3.3, and 3.4 on Linux:

And we test against 32- and 64-bit Python 2.7, 3.3, and 3.4 on Windows:

`filesystem_tree` is MIT-licensed.

CHAPTER 2

# Documentation

http://filesystem-tree-py.readthedocs.org/

## Tutorial

Instantiate the *FilesystemTree* class:

```
>>> from filesystem_tree import FilesystemTree
>>> ft = FilesystemTree()
```

Now make files!

```
>>> ft.mk(('greetings/program.txt', 'Greetings, program!'))
```

You can get the full absolute path of the file you created with *resolve*:

```
>>> filepath = ft.resolve('greetings/program.txt')
```

Make directories like so:

```
>>> ft.mk('my/directory/')
```

When you're done, clean up with *remove*:

```
>>> ft.remove()
```

Or use it as a context manager to clean up automatically:

```
>>> with FilesystemTree() as ft:
...     ft.mk('my/stuff')
```

# API Reference

**class** filesystem_tree.**FilesystemTree**(**treedef*, ***kw*)

Represent a filesystem tree.

**Parameters**

- **treedef** – Any positional arguments are passed through to *mk*.
- **root** (*string*) – The root of the filesystem tree. If not specified or None, a temporary directory will be created and used. (May only be supplied as a keyword argument.)
- **should_dedent** (*bool*) – Sets the instance default for whether or not the contents of files are dedented before being written. (May only be supplied as a keyword argument.)
- **encoding** (*str*) – Sets the instance default for what encoding to use when writing to disk. (May only be supplied as a keyword argument.)

Create a new instance of this class every time you need an isolated filesystem tree:

```
>>> ft = FilesystemTree()
```

This creates a temporary directory, the path to which you can access with ft.root:

```
>>> isdir(ft.root)
True
```

You can use it as a context manager to automatically *remove* the tree once you're done with it:

```
>>> with FilesystemTree() as ft:
...     pass
```

However, the tree is only removed if the code block doesn't raise an exception. If there's an exception the tree will be left on the filesystem so you can debug.

**mk**(**treedef*, ***kw*)

Builds a filesystem tree in root based on treedef.

**Parameters**

- **treedef** – The definition of a filesystem tree.
- **should_dedent** (*bool*) – Controls whether or not the contents of files are dedented before being written. If not specified, should_dedent is used. (May only be supplied as a keyword argument.)
- **encoding** (*str*) – The encoding with which to convert file contents to a bytestring if you specify said contents as a str (Python 3) or unicode (Python 2). If not specified, encoding is used. (May only be supplied as a keyword argument.)

**Raises** TypeError, if treedef contains anything besides strings and tuples; ValueError, if treedef contains a tuple that doesn't have two or three items

**Returns** None

This method iterates over the items in treedef, creating directories for any strings, and files for any tuples. For file tuples, the first item is the path of the file, the second is the contents to write, the third (optional) item is whether to dedent the contents first before writing it, and the fourth (optional) item is the encoding to use when writing the file. All paths must be specified using / as the separator (they will be automatically converted to the native path separator for the current platform). Any intermediate directories will be created as necessary.

So for example if you instantiate a *FilesystemTree*:

```
>>> ft = FilesystemTree()
```

And you call *mk* with:

```
>>> ft.mk(('path/to/file.txt', 'Greetings, program!'))
```

Then you'll have one file in your tree:

```
>>> files = os.listdir(os.path.join(ft.root, 'path', 'to'))
>>> print(' '.join(files))
file.txt
```

And it will have the content you asked for:

```
>>> open(ft.resolve('path/to/file.txt')).read()
'Greetings, program!'
```

The automatic dedenting is so you can use multi-line strings in indented code blocks to specify file contents and indent it with the rest of your code, but not have the indents actually written to the file. For example:

```
>>> def foo():
...     ft.mk(('other/file.txt', '''
...     Here is a list of things:
...         - Thing one.
...         - Thing two.
...         - Thing three.
...     '''))
...
>>> foo()
>>> print(open(ft.resolve('other/file.txt')).read())

Here is a list of things:
    - Thing one.
    - Thing two.
    - Thing three.
```

**remove**()

Remove the filesystem tree at `root`.

**Returns** `None`

**resolve**(*path=u''*)

Given a relative path, return an absolute path.

**Parameters** **path** – A path relative to `root` using `/` as the separator

**Returns** An absolute path using the native path separator, with symlinks removed

The return value of *resolve* with no arguments is equivalent to `root`.

# Python Module Index

## f

# Index