rpaths Documentation

Release 0.8

Remi Rampin

September 19, 2014

Contents

CHAPTER 1

Introduction

Python provides a module to manipulate filenames: `os.path`. However, it is very cumbersome to use (not object-oriented) and difficult to use safely (because of the bytes/unicode issue).

This library provides classes allowing you to perform all the common operations on paths easily, using ad-hoc classes.

Moreover, it aims at total Python 2/3 and Windows/POSIX interoperability. In every case, it will behave the "right way", even when dealing with POSIX filenames in broken unicode encodings.

CHAPTER 2

Classes

rpaths is organized in two levels. It offers abstract path representations, which only perform parsing/string manipulation and don't actually perform any operation on a file system. When dealing with abstract paths, nothing stops you from manipulating POSIX paths on a Windows host and vice-versa.

On top of these abstract paths comes the concrete `Path` class, which represents the native type for the current system. It inherits from the correct abstract class, and adds the actual system operations allowing you to resolve, list, create or remove files.

Note that, contrary to other path libraries, none of these types inherits from a built-in string class. However, you can build them from strings in a variety of ways and `repr()`, `bytes()` and `unicode()` will behave how you can expect.

2.1 Abstract classes

Abstract path behavior is defined by the `AbstractPath` class. You shouldn't use that directly, use `PosixPath` and `WindowsPath` which are its implementations.

class `rpaths.`**`AbstractPath`**`(`**parts*`)`

An abstract representation of a path.

This represents a path on a system that may not be the current one. It doesn't provide any way to actually interact with the local file system.

`ancestor``(`*n*`)`

Goes up n directories.

`components`

Splits this path into its components.

The first component will be the root if this path is relative, then each component leading to the filename.

`expand_user``()`

Replaces ~ or `~user` by that *user*'s home directory.

`expand_vars``()`

Expands environment variables in the path.

They might be of the form `$name` or `${name}`; references to non-existing variables are kept unchanged.

`ext`

The extension of this path.

`is_absolute`

Indicates whether this path is absolute or relative.

`lies_under`(*prefix*)
Indicates if the *prefix* is a parent of this path.

`name`
The name of this path, i.e. the final component without directories.

`norm_case`()
Removes the case if this flavor of paths is case insensitive.

`parent`
The parent directory of this path.

`rel_path_to`(*dest*)
Builds a relative path leading from this one to the given *dest*.

Note that these paths might be both relative, in which case they'll be assumed to start from the same directory.

`root`
The root of this path.

This will be either a root (with optionally a drive name or UNC share) or `'.'` for relative paths.

`split_root`()
Splits this path into a pair (drive, location).

Note that, because all paths are normalized, a root of '.' will be returned for relative paths.

`stem`
The name of this path without the extension.

`unicodename`
The name of this path as unicode.

2.2 Concrete class Path

The class Path represents a native path on your system. It inherits from either PosixPath or WindowsPath.

class `rpaths.`**`Path`**(**parts*)
A concrete representation of an actual path on this system.

This extends either `WindowsPath` or `PosixPath` depending on the current system. It adds concrete filesystem operations.

`absolute`()
Returns a normalized absolutized version of the pathname path.

`atime`()
Returns the time of last access to this path.

This returns a number of seconds since the epoch.

`chdir`()
Changes the current directory to this path.

`chmod`(*mode*)
Changes the mode of the path to the given numeric *mode*.

`chown`(*uid=-1*, *gid=-1*)
Changes the owner and group id of the path.

`copy`(*target*)
: Copies this file the *target*, which might be a directory.

  The permissions are copied.

`copyfile`(*target*)
: Copies this file to the given *target* location.

`copymode`(*target*)
: Copies the mode of this file on the *target* file.

  The owner is not copied.

`copystat`(*target*)
: Copies the permissions, times and flags from this to the *target*.

  The owner is not copied.

`copytree`(*target*, *symlinks=False*)
: Recursively copies this directory to the *target* location.

  The permissions and times are copied (like copystat).

  If the optional *symlinks* flag is true, symbolic links in the source tree result in symbolic links in the destination tree; if it is false, the contents of the files pointed to by symbolic links are copied.

`ctime`()
: Returns the ctime of this path.

  On some systems, this is the time of last metadata change, and on others (like Windows), it is the creation time for path. In any case, it is a number of seconds since the epoch.

classmethod `cwd`()
: Returns the current directory.

`exists`()
: True if the file exists, except for broken symlinks where it's False.

`hardlink`(*newpath*)
: Creates a hard link to this path at the given *newpath*.

`in_dir`(**args*, ***kwds*)
: Context manager that changes to this directory then changes back.

`is_dir`()
: True if this file exists and is a directory.

`is_file`()
: True if this file exists and is a regular file.

`is_link`()
: True if this file exists and is a symbolic link.

`is_mount`()
: True if this file is a mount point.

`lexists`()
: True if the file exists, even if it's a broken symbolic link.

`listdir`(*pattern=None*)
: Returns a list of all the files in this directory.

  The special entries `'.'` and `'..'` will not be returned.

mkdir(*name=None*, *parents=False*, *mode=511*)
Creates that directory, or a directory under this one.

`path.mkdir(name)` is a shortcut for `(path/name).mkdir()`.

move(*target*)
Recursively moves a file or directory to the given target location.

mtime()
Returns the time of last modification of this path.

This returns a number of seconds since the epoch.

open(*mode=u'r'*, *name=None*, ***kwargs*)
Opens this file, or a file under this directory.

`path.open(mode, name)` is a shortcut for `(path/name).open(mode)`.

Note that this uses `io.open()` which behaves differently from `open()` on Python 2; see the appropriate documentation.

read_link(*absolute=False*)
Returns the path this link points to.

If *absolute* is True, the target is made absolute.

recursedir(*pattern=None*, *top_down=True*)
Recursively lists all files under this directory.

Symbolic links will be walked but files will never be duplicated.

This accepts extended patterns, where:

- a slash '/' always represents the path separator
- a backslash '' escapes other special characters
- an initial slash '/' anchors the match at the beginning of the (relative) path
- a trailing '/' suffix is removed
- an asterisk '*' matches a sequence of any length (including 0) of any characters (except the path separator)
- a '?' matches exactly one character (except the path separator)
- '[abc]' matches characters 'a', 'b' or 'c'
- two asterisks '**' matches one or more path components (might match '/' characters)

rel_path_to(*dest*)
Builds a relative path leading from this one to another.

Note that these paths might be both relative, in which case they'll be assumed to be considered starting from the same directory.

Contrary to `AbstractPath`'s version, this will also work if one path is relative and the other absolute.

relative()
Builds a relative version of this path from the current directory.

This is the same as `Path.cwd().rel_path_to(thispath)`.

remove()
Removes this file.

rename (*new*, *parents=False*)
Renames this path to the given new location.

If *parents* is True, it will create the parent directories of the target if they don't exist.

resolve ()
Expands the symbolic links in the path.

rewrite (**args*, ***kwds*)
Replaces this file with new content.

> This context manager gives you two file objects, (r, w), where r is readable and has the current content of the file, and w is writable and will replace the file at the end of the context (unless an exception is raised, in which case it is rolled back).
>
> Keyword arguments will be used for both files, unless they are prefixed with `read_` or `write_`. For instance:
>
> ```
> with Path('test.txt').rewrite(read_newline='
> ```

', write_newline='

') as (r, w): w.write(r.read())

rmdir (*parents=False*)
Removes this directory, provided it is empty.

Use `rmtree()` if it might still contain files.

If parents is True, it will also destroy every empty directory above it until an error is encountered.

rmtree (*ignore_errors=False*)
Deletes an entire directory.

If ignore_errors is True, failed removals will be ignored; else, an exception will be raised.

same_file (*other*)
Returns True if both paths refer to the same file or directory.

In particular, this identifies hard links.

size ()
Returns the size, in bytes, of the file.

symlink (*target*)
Create a symbolic link here, pointing to the given *target*.

classmethod tempdir (*suffix=u''*, *prefix=None*, *dir=None*)
Returns a new temporary directory.

Arguments are as for tempfile, except that the *text* argument is not accepted.

The directory is readable, writable, and searchable only by the creating user.

Caller is responsible for deleting the directory when done with it.

classmethod tempfile (*suffix=u''*, *prefix=None*, *dir=None*, *text=False*)
Returns a new temporary file.

The return value is a pair (fd, path) where fd is the file descriptor returned by `os.open()`, and path is a `Path` to it.

If *suffix* is specified, the file name will end with that suffix, otherwise there will be no suffix.

If *prefix* is specified, the file name will begin with that prefix, otherwise a default prefix is used.

If *dir* is specified, the file will be created in that directory, otherwise a default directory is used.

If *text* is specified and true, the file is opened in text mode. Else (the default) the file is opened in binary mode. On some operating systems, this makes no difference.

The file is readable and writable only by the creating user ID. If the operating system uses permission bits to indicate whether a file is executable, the file is executable by no one. The file descriptor is not inherited by children of this process.

The caller is responsible for deleting the file when done with it.