pathlib — Filesystem Paths as Objects

Purpose: Parse, build, test, and otherwise work on filenames and paths using an object-oriented API instead of low-level string operations.

Path Representations

`pathlib` includes classes for managing filesystem paths formatted using either the POSIX standard or Microsoft Windows syntax. It includes so called “pure” classes, which operate on strings but do not interact with an actual filesystem, and “concrete” classes, which extend the API to include operations that reflect or modify data on the local filesystem.

The pure classes `PurePosixPath` and `PureWindowsPath` can be instantiated and used on any operating system, since they only work on names. To instantiate the correct class for working with a real filesystem, use `Path` to get either a `PosixPath` or `WindowsPath`, depending on the platform.

Building Paths

To instantiate a new path, give a string as the first argument. The string representation of the path object is this name value. To create a new path referring to a value relative to an existing path, use the `/` operator to extend the path. The argument to the operator can either be a string or another path object.

```
# pathlib_operator.py

import pathlib

usr = pathlib.PurePosixPath('/usr')
print(usr)

usr_local = usr / 'local'
print(usr_local)

usr_share = usr / pathlib.PurePosixPath('share')
print(usr_share)

root = usr / '..'
print(root)

etc = root / '/etc/'
print(etc)
```

As the value for `root` in the example output shows, the operator combines the path values as they are given, and does not normalize the result when it contains the parent directory reference `".."`. However, if a segment begins with the path separator it is interpreted as a new “root” reference in the same way as `os.path.join()`. Extra path separators are removed from the middle of the path value, as in the `etc` example here.

```
$ python3 pathlib_operator.py

/usr
/usr/local
/usr/share
/usr/..
/etc
```

The concrete path classes include a `resolve()` method for normalizing a path by looking at the filesystem for directories and symbolic links and producing the absolute path referred to by a name.

```
# pathlib_resolve.py

import pathlib

usr_local = pathlib.Path('/usr/local')
share = usr local / '..' / 'share'
```

```
print(share.resolve())
```

Here the relative path is converted to the absolute path to `/usr/share`. If the input path includes symlinks, those are expanded as well to allow the resolved path to refer directly to the target.

```
$ python3 pathlib_resolve.py

/usr/share
```

To build paths when the segments are not known in advance, use `joinpath()`, passing each path segment as a separate argument.

```
# pathlib_joinpath.py

import pathlib

root = pathlib.PurePosixPath('/')
subdirs = ['usr', 'local']
usr_local = root.joinpath(*subdirs)
print(usr_local)
```

As with the `/` operator, calling `joinpath()` creates a new instance.

```
$ python3 pathlib_joinpath.py

/usr/local
```

Given an existing path object, it is easy to build a new one with minor differences such as referring to a different file in the same directory. Use `with_name()` to create a new path that replaces the name portion of a path with a different file name. Use `with_suffix()` to create a new path that replaces the file name's extension with a different value.

```
# pathlib_from_existing.py

import pathlib

ind = pathlib.PurePosixPath('source/pathlib/index.rst')
print(ind)

py = ind.with_name('pathlib_from_existing.py')
print(py)

pyc = py.with_suffix('.pyc')
print(pyc)
```

Both methods return new objects, and the original is left unchanged.

```
$ python3 pathlib_from_existing.py

source/pathlib/index.rst
source/pathlib/pathlib_from_existing.py
source/pathlib/pathlib_from_existing.pyc
```

Parsing Paths

Path objects have methods and properties for extracting partial values from the name. For example, the `parts` property produces a sequence of path segments parsed based on the path separator value.

```
# pathlib_parts.py

import pathlib

p = pathlib.PurePosixPath('/usr/local')
print(p.parts)
```

The sequence is a tuple, reflecting the immutability of the path instance.

```
$ python3 pathlib_parts.py

('/', 'usr', 'local')
```

There are two ways to navigate “up” the filesystem hierarchy from a given path object. The `parent` property refers to a new path instance for the directory containing the path, the value returned by `os.path.dirname()`. The `parents` property is an iterable that produces parent directory references, continually going “up” the path hierarchy until reaching the root.

```
# pathlib_parents.py

import pathlib

p = pathlib.PurePosixPath('/usr/local/lib')

print('parent: {}'.format(p.parent))

print('\nhierarchy:')
for up in p.parents:
    print(up)
```

The example iterates over the `parents` property and prints the member values.

```
$ python3 pathlib_parents.py

parent: /usr/local

hierarchy:
/usr/local
/usr
/
```

Other parts of the path can be accessed through properties of the path object. The `name` property holds the last part of the path, after the final path separator (the same value that `os.path.basename()` produces). The `suffix` property holds the value after the extension separator and the `stem` property holds the portion of the name before the suffix.

```
# pathlib_name.py

import pathlib

p = pathlib.PurePosixPath('./source/pathlib/pathlib_name.py')
print('path  : {}'.format(p))
print('name  : {}'.format(p.name))
print('suffix: {}'.format(p.suffix))
print('stem  : {}'.format(p.stem))
```

Although the `suffix` and `stem` values are similar to the values produced by `os.path.splitext()`, the values are based only on the value of `name` and not the full path.

```
$ python3 pathlib_name.py

path  : source/pathlib/pathlib_name.py
name  : pathlib_name.py
suffix: .py
stem  : pathlib_name
```

Creating Concrete Paths

Instances of the concrete `Path` class can be created from string arguments referring to the name (or potential name) of a file, directory, or symbolic link on the file system. The class also provides several convenience methods for building instances using commonly used locations that change, such as the current working directory and the user’s home directory.

```
# pathlib_convenience.py

import pathlib

home = pathlib.Path.home()
print('home: ', home)
```

```
cwd = pathlib.Path.cwd()
print('cwd : ', cwd)
```

Both methods create Path instances pre-populated with an absolute file system reference.

```
$ python3 pathlib_convenience.py

home:  /Users/dhellmann
cwd :  /Users/dhellmann/PyMOTW
```

Directory Contents

There are three methods for accessing the directory listings to discover the names of files available on the file system. `iterdir()` is a generator, yielding a new Path instance for each item in the containing directory.

```
# pathlib_iterdir.py

import pathlib

p = pathlib.Path('.')

for f in p.iterdir():
    print(f)
```

If the Path does not refer to a directory, `iterdir()` raises `NotADirectoryError`.

```
$ python3 pathlib_iterdir.py

example_link
index.rst
pathlib_chmod.py
pathlib_convenience.py
pathlib_from_existing.py
pathlib_glob.py
pathlib_iterdir.py
pathlib_joinpath.py
pathlib_mkdir.py
pathlib_name.py
pathlib_operator.py
pathlib_ownership.py
pathlib_parents.py
pathlib_parts.py
pathlib_read_write.py
pathlib_resolve.py
pathlib_rglob.py
pathlib_rmdir.py
pathlib_stat.py
pathlib_symlink_to.py
pathlib_touch.py
pathlib_types.py
pathlib_unlink.py
```

Use `glob()` to find only files matching a pattern.

```
# pathlib_glob.py

import pathlib

p = pathlib.Path('..')

for f in p.glob('*.rst'):
    print(f)
```

This example shows all of the reStructuredText input files in the parent directory of the script.

```
$ python3 pathlib_glob.py
```

```
../about.rst
../algorithm_tools.rst
../book.rst
../compression.rst
../concurrency.rst
../cryptographic.rst
../data_structures.rst
../dates.rst
../dev_tools.rst
../email.rst
../file_access.rst
../frameworks.rst
../i18n.rst
../importing.rst
../index.rst
../internet_protocols.rst
../language.rst
../networking.rst
../numeric.rst
../persistence.rst
../porting_notes.rst
../runtime_services.rst
../text.rst
../third_party.rst
../unix.rst
```

The glob processor supports recursive scanning using the pattern prefix ** or by calling `rglob()` instead of `glob()`.

```
# pathlib_rglob.py

import pathlib

p = pathlib.Path('..')

for f in p.rglob('pathlib_*.py'):
    print(f)
```

Because this example starts from the parent directory, a recursive search is necessary to find the example files matching `pathlib_*.py`.

```
$ python3 pathlib_rglob.py

../pathlib/pathlib_chmod.py
../pathlib/pathlib_convenience.py
../pathlib/pathlib_from_existing.py
../pathlib/pathlib_glob.py
../pathlib/pathlib_iterdir.py
../pathlib/pathlib_joinpath.py
../pathlib/pathlib_mkdir.py
../pathlib/pathlib_name.py
../pathlib/pathlib_operator.py
../pathlib/pathlib_ownership.py
../pathlib/pathlib_parents.py
../pathlib/pathlib_parts.py
../pathlib/pathlib_read_write.py
../pathlib/pathlib_resolve.py
../pathlib/pathlib_rglob.py
../pathlib/pathlib_rmdir.py
../pathlib/pathlib_stat.py
../pathlib/pathlib_symlink_to.py
../pathlib/pathlib_touch.py
../pathlib/pathlib_types.py
../pathlib/pathlib_unlink.py
```

Reading and Writing Files

Each Path instance includes methods for working with the contents of the file to which it refers. For immediately retrieving the contents, use `read_bytes()` or `read_text()`. To write to the file, use `write_bytes()` or `write_text()`. Use the `open()` method to open the file and retain the file handle, instead of passing the name to the built-in `open()` function.

```
# pathlib_read_write.py

import pathlib

f = pathlib.Path('example.txt')

f.write_bytes('This is the content'.encode('utf-8'))

with f.open('r', encoding='utf-8') as handle:
    print('read from open(): {!r}'.format(handle.read()))

print('read_text(): {!r}'.format(f.read_text('utf-8')))
```

The convenience methods do some type checking before opening the file and writing to it, but otherwise they are equivalent to doing the operation directly.

```
$ python3 pathlib_read_write.py

read from open(): 'This is the content'
read_text(): 'This is the content'
```

Manipulating Directories and Symbolic Links

Paths representing directories or symbolic links that do not exist can be used to create the associated file system entries.

```
# pathlib_mkdir.py

import pathlib

p = pathlib.Path('example_dir')

print('Creating {}'.format(p))
p.mkdir()
```

If the path already exists, `mkdir()` raises a `FileExistsError`.

```
$ python3 pathlib_mkdir.py

Creating example_dir

$ python3 pathlib_mkdir.py

Creating example_dir
Traceback (most recent call last):
  File "pathlib_mkdir.py", line 16, in <module>
    p.mkdir()
  File ".../lib/python3.6/pathlib.py", line 1226, in mkdir
    self._accessor.mkdir(self, mode)
  File ".../lib/python3.6/pathlib.py", line 387, in wrapped
    return strfunc(str(pathobj), *args)
FileExistsError: [Errno 17] File exists: 'example_dir'
```

Use `symlink_to()` to create a symbolic link. The link will be named based on the path's value and will refer to the name given as argument to `symlink_to()`.

```
# pathlib_symlink_to.py

import pathlib

p = pathlib.Path('example_link')

p.symlink_to('index.rst')

print(p)
print(p.resolve().name)
```

This example creates a symbolic link, then uses `resolve()` to read the link to find what it points to and print the name.

```
$ python3 pathlib_symlink_to.py

example_link
index.rst
```

File Types

A Path instance includes several methods for testing the type of file refered to by the path. This example creates several files of different types and tests those as well as a few other device-specific files available on the local operating system.

```
# pathlib_types.py
```

```
import itertools
import os
import pathlib

root = pathlib.Path('test_files')

# Clean up from previous runs.
if root.exists():
    for f in root.iterdir():
        f.unlink()
else:
    root.mkdir()

# Create test files
(root / 'file').write_text(
    'This is a regular file', encoding='utf-8')
(root / 'symlink').symlink_to('file')
os.mkfifo(str(root / 'fifo'))

# Check the file types
to_scan = itertools.chain(
    root.iterdir(),
    [pathlib.Path('/dev/disk0'),
     pathlib.Path('/dev/console')],
)
hfmt = '{:18s}' + ('  {:>5}' * 6)
print(hfmt.format('Name', 'File', 'Dir', 'Link', 'FIFO', 'Block',
                  'Character'))
print()

fmt = '{:20s}  ' + ('{!r:>5}  ' * 6)
for f in to_scan:
    print(fmt.format(
        str(f),
        f.is_file(),
        f.is_dir(),
        f.is_symlink(),
        f.is_fifo(),
        f.is_block_device(),
        f.is_char_device(),
    ))
```

Each of the methods, `is_dir()`, `is_file()`, `is_symlink()`, `is_socket()`, `is_fifo()`, `is_block_device()`, and `is_char_device()`, takes no arguments.

```
$ python3 pathlib_types.py

Name                 File    Dir   Link   FIFO  Block  Character

test_files/fifo       False  False  False   True  False  False
test_files/file        True  False  False  False  False  False
test_files/symlink     True  False   True  False  False  False
/dev/disk0            False  False  False  False   True  False
/dev/console          False  False  False  False  False   True
```

File Properties

File Properties

Detailed information about a file can be accessed using the methods `stat()` or `lstat()` (for checking the status of something that might be a symbolic link). These methods produce the same results as `os.stat()` and `os.lstat()`.

```
# pathlib_stat.py

import pathlib
import sys
import time

if len(sys.argv) == 1:
    filename = __file__
else:
    filename = sys.argv[1]

p = pathlib.Path(filename)
stat_info = p.stat()

print('{}:'.format(filename))
print('  Size:', stat_info.st_size)
print('  Permissions:', oct(stat_info.st_mode))
print('  Owner:', stat_info.st_uid)
print('  Device:', stat_info.st_dev)
print('  Created      :', time.ctime(stat_info.st_ctime))
print('  Last modified:', time.ctime(stat_info.st_mtime))
print('  Last accessed:', time.ctime(stat_info.st_atime))
```

The output will vary depending on how the example code was installed. Try passing different filenames on the command line to `pathlib_stat.py`.

```
$ python3 pathlib_stat.py

pathlib_stat.py:
  Size: 607
  Permissions: 0o100644
  Owner: 527
  Device: 16777220
  Created      : Thu Dec 29 12:38:23 2016
  Last modified: Thu Dec 29 12:38:23 2016
  Last accessed: Sun Mar 18 16:21:41 2018

$ python3 pathlib_stat.py index.rst

index.rst:
  Size: 19569
  Permissions: 0o100644
  Owner: 527
  Device: 16777220
  Created      : Sun Mar 18 16:11:31 2018
  Last modified: Sun Mar 18 16:11:31 2018
  Last accessed: Sun Mar 18 16:21:40 2018
```

For simpler access to information about the owner of a file, use `owner()` and `group()`.

```
# pathlib_ownership.py

import pathlib

p = pathlib.Path(__file__)

print('{} is owned by {}/{}'.format(p, p.owner(), p.group()))
```

While `stat()` returns numerical system ID values, these methods look up the name associated with the IDs.

```
$ python3 pathlib_ownership.py

pathlib_ownership.py is owned by dhellmann/dhellmann
```

The `touch()` method works like the Unix command `touch` to create a file or update an existing file's modification time and permissions.

```
# pathlib_touch.py

import pathlib
import time

p = pathlib.Path('touched')
if p.exists():
    print('already exists')
else:
    print('creating new')

p.touch()
start = p.stat()

time.sleep(1)

p.touch()
end = p.stat()

print('Start:', time.ctime(start.st_mtime))
print('End  :', time.ctime(end.st_mtime))
```

Running this example more than once updates the existing file on subsequent runs.

```
$ python3 pathlib_touch.py

creating new
Start: Sun Mar 18 16:21:41 2018
End  : Sun Mar 18 16:21:42 2018

$ python3 pathlib_touch.py

already exists
Start: Sun Mar 18 16:21:42 2018
End  : Sun Mar 18 16:21:43 2018
```

Permissions

On Unix-like systems, file permissions can be changed using `chmod()`, passing the mode as an integer. Mode values can be constructed using constants defined in the `stat` module. This example toggles the user's execute permission bit.

```
# pathlib_chmod.py

import os
import pathlib
import stat

# Create a fresh test file.
f = pathlib.Path('pathlib_chmod_example.txt')
if f.exists():
    f.unlink()
f.write_text('contents')

# Determine what permissions are already set using stat.
existing_permissions = stat.S_IMODE(f.stat().st_mode)
print('Before: {:o}'.format(existing_permissions))

# Decide which way to toggle them.
if not (existing_permissions & os.X_OK):
    print('Adding execute permission')
    new_permissions = existing_permissions | stat.S_IXUSR
else:
    print('Removing execute permission')
    # use xor to remove the user execute permission
    new_permissions = existing_permissions ^ stat.S_IXUSR
```

```
# Make the change and show the new value.
f.chmod(new_permissions)
after_permissions = stat.S_IMODE(f.stat().st_mode)
print('After: {:o}'.format(after_permissions))
```

The script assumes it has the permissions necessary to modify the mode of the file when run.

```
$ python3 pathlib_chmod.py

Before: 644
Adding execute permission
After: 744
```

Deleting

There are two methods for removing things from the file system, depending on the type. To remove an empty directory, use `rmdir()`.

```
# pathlib_rmdir.py

import pathlib

p = pathlib.Path('example_dir')

print('Removing {}'.format(p))
p.rmdir()
```

A `FileNotFoundError` exception is raised if the post-conditions are already met and the directory does not exist. It is also an error to try to remove a directory that is not empty.

```
$ python3 pathlib_rmdir.py

Removing example_dir

$ python3 pathlib_rmdir.py

Removing example_dir
Traceback (most recent call last):
  File "pathlib_rmdir.py", line 16, in <module>
    p.rmdir()
  File ".../lib/python3.6/pathlib.py", line 1270, in rmdir
    self._accessor.rmdir(self)
  File ".../lib/python3.6/pathlib.py", line 387, in wrapped
    return strfunc(str(pathobj), *args)
FileNotFoundError: [Errno 2] No such file or directory:
'example_dir'
```

For files, symbolic links, and most other path types use `unlink()`.

```
# pathlib_unlink.py

import pathlib

p = pathlib.Path('touched')

p.touch()

print('exists before removing:', p.exists())

p.unlink()

print('exists after removing:', p.exists())
```

The user must have permission to remove the file, symbolic link, socket, or other file system object.

```
$ python3 pathlib_unlink.py

exists before removing: True
```

```
exists before removing: True
exists after removing: False
```

See also

- Standard library documentation for pathlib
- `os.path` – Platform-independent manipulation of filenames
- Managing File System Permissions – Discussion of `os.stat()` and `os.lstat()`.
- `glob` – Unix shell pattern matching for filenames
- **PEP 428** – The pathlib module

os.path — Platform-independent Manipulation of Filenames

glob — Filename Pattern Matching

This page was last updated 2018-03-18.

Navigation

os.path — Platform-independent Manipulation of Filenames
glob — Filename Pattern Matching

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

- Blog
- The Python Standard Library By Example