COLUMNS

Revisiting Pathlib

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

Over the last few years Python has changed substantially, introducing a variety of new language syntax and libraries. While certain features have received more of the limelight (e.g., asynchronous I/O), an easily overlooked aspect of Python is its revamped handling of file names and directories. I introduced some of this when I wrote about the `pathlib` module in *;login:* back in October 2014 [1]). Since writing that, however, I've been unable to bring myself to use this new feature of the library. It was simply too different, and it didn't play nicely with others. Apparently, I wasn't alone in finding it strange--`pathlib` [2] was almost removed from the standard library before being rescued in Python 3.6. Given that three years have passed, maybe it's time to revisit the topic of file and directory handling.

The Old Ways

If you have to do anything with files and directories, you know that the functionality is spread out across a wide variety of built-in functions and standard library modules. For example, you have the `open` function for opening files:

```
with open('Data.txt') as f:
    data = f.read()
```

And there are functions in the `os` module for dealing with directories:

```
import os

files = os.listdir('.')     # Directory listing
os.mkdir('data')            # Make a directory
```

And then there is the problem of manipulating pathnames. For that, there is the `os.path` module. For example, if you needed to pull a file name apart, you could write code like this:

```
>>> filename = '/Users/beazley/Pictures/img123.jpg'
>>> import os.path

>>> # Get the base directory name
>>> os.path.dirname(filename)
'/Users/beazley/Pictures'

>>> # Get the base filename
>>> os.path.basename(filename)
'img123.jpg'

>>> # Split a filename into directory and filename components
>>> os.path.split(filename)
('/Users/beazley/Pictures', 'img123.jpg')
```

```
>>> # Get the filename and extension
>>> os.path.splitext(filename)
('/Users/beazley/Pictures/img123', '.jpg')
>>>

>>> # Get just the extension
>>> os.path.splitext(filename)[1]
'.jpg'
>>>
```

Or if you needed to rewrite part of a file name, you might do this:

```
>>> filename
'/Users/beazley/Pictures/img123.jpg'
>>> dirname, basename = os.path.split(filename)
>>> base, ext = os.path.splitext(basename)
>>> newfilename = os.path.join(dirname, 'thumbnails', base+'.png')
>>> newfilename
'/Users/beazley/Pictures/thumbnails/img123.png'
>>>
```

Finally, there are an assortment of other file-related features that get regular use. For example, the `glob` module can be used to get file listings with shell wildcards. The `shutil` module has functions for copying and moving files. The `os` module has a `walk()` function for walking directories. You might use these to search for files and perform some kind of processing:

```
import os
import os.path
import glob

def make_dir_thumbnails(dirname, pat):
    filenames = glob.glob(os.path.join(dirname, pat))
    for filename in filenames:
       dirname, basename = os.path.split(filename)
       base, ext = os.path.splitext(basename)
       origfilename = os.path.join(dirname, filename)
       newfilename = os.path.join(dirname, 'thumbnails', base+'.png')
       print('Making thumbnail %s -> %s' % (filename, newfilename))
       out = subprocess.check_output(['convert', '-resize',
          '100x100', origfilename, newfilename])

def make_all_thumbnails(dirname, pat):
    for path, dirs, files in os.walk(dirname):
        make_dir_thumbnails(path, pat)

# Example
make_all_thumbnails('/Users/beazley/PhotoLibrary', '*.JPG')
```

If you've written any kind of Python code that manipulates files, you're probably already familiar with this sort of code (for better or worse).

The New Way

Starting in Python 3.4, it became possible to think about pathnames in a different way. Instead of merely being a string, a pathname could be a proper object in its own right. For example, you could make a Path [3] instance and do this like this:

```
>>> from pathlib import Path
>>> filename = Path('/Users/beazley/Pictures/img123.jpg')
>>> filename
PosixPath('/Users/beazley/Pictures/img123.jpg')
>>> data = filename.read_bytes()
>>> newname = filename.with_name('backup_' + filename.name)
>>> newname
PosixPath('/Users/beazley/Pictures/backup_img123.jpg')
>>> newname.write_bytes(data)
1805312
>>>
```

Manipulation of the file name itself turns into methods:

```
>>> filename.parent
PosixPath('/Users/beazley/Pictures')
>>> filename.name
'img123.jpg'
>>> filename.parts
('/', 'Users', 'beazley', 'Pictures', 'img123.jpg')
>>> filename.parent / 'newdir' / filename.name
PosixPath('/Users/beazley/Pictures/newdir/img123.jpg')
>>> filename.stem
'img123'
>>> filename.suffix
'.jpg'
>>> filename.with_suffix('.png')
PosixPath('/Users/beazley/Pictures/img123.png')
>>> filename.as_uri()
'file:///Users/beazley/Pictures/img123.jpg'
>>> filename.match('*.jpg')
True
>>>
```

Paths have a lot of other useful features. For example, you can easily get file metadata:

```
>>> filename.exists()
True
>>> filename.is_file()
True
>>> filename.owner()
'beazley'
>>> filename.stat().st_size
1805312
>>> filename.stat().st_mtime
1388575451
>>>
```

There are also some nice directory manipulation features. For example, the `glob` method returns an iterator for finding matching files:

```
>>> pics = Path('/Users/beazley/Pictures')
>>> for pngfile in pics.glob('*.PNG'):
...     print(pngfile)
...
/Users/beazley/Pictures/IMG_3383.PNG
/Users/beazley/Pictures/IMG_3384.PNG
/Users/beazley/Pictures/IMG_3385.PNG
...
>>>
```

If you use `rglob()`, you will search an entire directory tree. For example, this finds all PNG files in my home directory:

```
for pngfile in Path('/Users/beazley').rglob('*.PNG'):
      print(pngfile)
```

The Achilles Heel…And Much Sadness

At first glance, it looks like `Path` objects are quite useful—and they are. Until recently, however, they were a bit of an "all-in" proposition: if you created a `Path` object, it couldn't be used anywhere else in the non-path world. In Python 3.5, for example, you'd get all sorts of errors if you ever used a `Path` with more traditional file-related functionality:

```
>>> # PYTHON 3.5
>>> filename = Path('/Users/beazley/Pictures/img123.png')
>>> open(filename, 'rb')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: invalid file: PosixPath('/Users/beazley/Pictures/
img123.png')

>>> os.path.dirname(filename)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.5/posixpath.py", line 148,
in dirname i = p.rfind(sep) + 1
AttributeError: 'PosixPath' object has no attribute 'rfind'
>>>

>>> import subprocess
>>> subprocess.check_output(['convert', '-resize', '100x100',
filename, newfilename])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.5/subprocess.py", line 626,
in check_output **kwargs).stdout
  File "/usr/local/lib/python3.5/subprocess.py", line 693,
in run with Popen(*popenargs, **kwargs) as process:
  File "/usr/local/lib/python3.5/subprocess.py", line 947,
in __init__ restore_signals, start_new_session)
  File "/usr/local/lib/python3.5/subprocess.py", line 1490,
in _execute_child restore_signals, start_new_session, preexec_fn)
TypeError: Can't convert 'PosixPath' object to str implicitly
>>>
```

Basically, `pathlib` partitioned Python into two worlds—the world of `pathlib` and the world of everything else. It's not entirely unlike the separation of Unicode versus bytes, which is to say rather unpleasant if you don't know what's going on. You could get around these limitations, but the fix involves placing explicit string conversions everywhere. For example:

```
>>> import subprocess
>>> subprocess.check_output(['convert', '-resize', '100x100',
str(filename), str(newfilename)])
>>>
```

Frankly, that's pretty annoying. It makes it virtually impossible to pass `Path` objects around in your program as a substitute for a file name. Everywhere that passed the name a low-level function would have to remember to include the string conversion. Modifying the whole universe of Python code is just not practical. It's forcing me to think about a problem that I don't want to think about.

Python 3.6 to the Rescue!

The good news is that `pathlib` was rescued in Python 3.6. A new magic protocol was introduced for file names. Specifically, if a class defines a `__fspath__()` method, it is called to produce a valid path string. For example:

```
>>> filename = Path('/Users/beazley/Pictures/img123.png')
>>> filename.__fspath__()
'/Users/beazley/Pictures/img123.png'
>>>
```

A corresponding function `fspath()` that was added to the `os` module for coercing a path to a string (or returning a string unmodified):

```
>>> import os
>>> os.fspath(filename)
'/Users/beazley/Pictures/img123.png'
>>>
```

A corresponding C API function was also added so that C extensions to Python could receive path-like objects.

Finally, there is also an abstract base class that can be used to implement your own custom path objects:

```
class MyPath(os.PathLike):
    def __init__(self, name):
        self.name = name
    def __fspath__(self):
        print('Converting path')
        return self.name
```

The above class allows you to investigate conversions. For example:

```
>>> p = MyPath('/Users/beazley/Pictures/img123.jpg')
>>> f = open(p, 'rb')
Converting path
>>> os.path.dirname(p)
Converting path
'/Users/beazley/Pictures'
>>> subprocess.check_output(['ls', p])
Converting path
b'/Users/beazley/Pictures/img123.png\n'
>>>
```

So far as I can tell, the integration of Path objects with the Python standard library is fairly extensive. All of the core file-related functionality in modules such as os, os.path, shutil, subprocess seems to work. By extension, nearly any standard library module that accepts a file name and uses that standard functionality will also work. It's nice. Here's a revised example of code that uses pathlib:

```
from pathlib import Path
import subprocess
def make_thumbnails(topdir, pat):
    topdir = Path(topdir)
    for filename in topdir.rglob(pat):
        newdirname = filename.parent / 'thumbnails'
        newdirname.mkdir(exist_ok=True)
        newfilename = newdirname / (filename.stem + '.png')
        out = subprocess.check_output(['convert', '-resize','100x100',
                                        filename, newfilename])
if __name__ == '__main__':
    make_thumbnails('/Users/beazley/PhotoLibrary', '*.JPG')
```

That's pretty nice.

Potential Potholes

Alas, all is still not entirely perfect in the world of paths. One area where you could get tripped up is in code that's too finicky about type checking. For example, a function like this will hate paths:

```
def read_data(filename):
    assert isinstance(filename, str), "Filename must be a string"
    ...
```

If you're a library writer, it's probably best to coerce the input through os.fspath() instead. This will report an exception if the input isn't compatible. Thus, you could write this:

```
def read_data(filename):
    filename = os.fspath(filename)
    ...
```

You can also get tripped up by code that assumes the use of strings and performs string manipulation to do things with file names. For example:

```
def make_backup(filename):
    backup_file = filename + '.bak'
    ...
```

If you pass a Path object to this function, it will crash with a TypeError since Path instances don't implement the + operator. Shame on the author for not using the os.path module in the first place. Again, the problem can likely be solved with a coercion.

```
def make_backup(filename):
    filename = os.fspath(filename)
    backup_file = filename + '.bak'
    ...
```

But be aware that file names are allowed to be byte-strings. Even if you make the above change, the code is still basically broken. The concatenation will fail if a byte-string file name is passed.

C extensions accepting file names could also potentially break unless they are using the new protocol. Hopefully, such cases are rare—it's not too common to see libraries that directly open files on their own as opposed to using Python's built-in functions.

Final Words

All things considered, it now seems like pathlib might be something that can be used as a replacement for os.path without too much annoyance. Now, I just need to train my brain to use it—honestly, this might be even harder than switching from print to print(). However, let's not discuss that.

References

[1] D. Beazley, "A Path Less Traveled," *;login:*, vol. 39, no. 5 (October 2014), pp. 47–51: https://www.usenix.org/system/files/login/articles/login_1410_10_beazley.pdf.

[2] pathlib module: https://docs.python.org/3/library/pathlib.html.

[3] PEP 519—Adding a file system path protocol: https://www.python.org/dev/peps/pep-0519/.