Trey Hunner

I help developers level-up their Python skills

Hire Me For Training

- RSS

Search

Navigate…

- Articles
- Talks
- Python Morsels
- Team Training
- About

Why you should be using pathlib

Dec 21st, 2018 2:00 pm | Comments

When I discovered Python's new pathlib module a few years ago, I initially wrote it off as being a slightly more awkward and unnecessarily object-oriented version of the `os.path` module. I was wrong. Python's `pathlib` module is actually wonderful!

In this article I'm going to try to sell you on `pathlib`. I hope that this article will inspire you to **use Python's `pathlib` module pretty much anytime you need to work with files in Python**.

Update: I wrote a follow-up article to address further comments and concerns that were raised after this one. Read this article first and then take a look at the follow-up article here.

os.path is clunky

The `os.path` module has always been what we reached for to work with paths in Python. It's got pretty much all you need, but it can be very clunky sometimes.

Should you import it like this?

```
import os.path

BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
TEMPLATES_DIR = os.path.join(BASE_DIR, 'templates')
```

Or like this?

```
from os.path import abspath, dirname, join

BASE_DIR = dirname(dirname(abspath(__file__)))
TEMPLATES_DIR = join(BASE_DIR, 'templates')
```

Or maybe that `join` function is too generically named… so we could do this instead:

```
from os.path import abspath, dirname, join as joinpath

BASE_DIR = dirname(dirname(abspath(__file__)))
TEMPLATES_DIR = joinpath(BASE_DIR, 'templates')
```

I find all of these a bit awkward. We're passing strings into functions that return strings which we then pass into other functions that return strings. All of these strings happen to represent paths, but they're still just strings.

The string-in-string-out functions in `os.path` are really awkward when nested because the code has to be read from the inside out. Wouldn't it be nice if we could take these nested function calls and turn them into chained method calls instead?

With the `pathlib` module we can!

```
1 from pathlib import Path
2
3 BASE_DIR = Path(__file__).resolve().parent.parent
4 TEMPLATES_DIR = BASE_DIR.joinpath('templates')
```

The `os.path` module requires function nesting, but **the `pathlib` modules' `Path` class allows us to chain methods and attributes** on `Path` objects to get an equivalent path representation.

I know what you're thinking: wait these `Path` objects aren't the same thing: they're objects, not path strings! I'll address that later (hint: these can pretty much be used interchangeably with path strings).

The os module is crowded

Python's classic `os.path` module is just for working with paths. Once you want to actually *do* something with a path (e.g. create a directory) you'll need to reach for another Python module, often the `os` module.

The `os` module has lots of utilities for working with files and directories: `mkdir`, `getcwd`, `chmod`, `stat`, `remove`, `rename`, and `rmdir`. Also `chdir`, `link`, `walk`, `listdir`, `makedirs`, `renames`, `removedirs`, `unlink` (same as `remove`), and `symlink`. And a bunch of other stuff that isn't related to the filesystems at all: `fork`, `getenv`, `putenv`, `environ`, `getlogin`, and `system`. Plus dozens of things I didn't mention in this paragraph.

Python's `os` module does a little bit of everything; it's sort of a junk drawer for system-related stuff. There's a lot of lovely stuff in the `os` module, but it can be hard to find what you're looking for sometimes: if you're looking for path-related or filesystem-related things in the `os` module, you'll need to do a bit of digging.

The `pathlib` module replaces many of these filesystem-related `os` utilities with methods on the `Path` object.

Here's some code that makes a `src/__pypackages__` directory and renames our `.editorconfig` file to `src/.editorconfig`:

```
1 import os
2 import os.path
3
4 os.makedirs(os.path.join('src', '__pypackages__'), exist_ok=True)
5 os.rename('.editorconfig', os.path.join('src', '.editorconfig'))
```

This code does the same thing using `Path` objects:

```
1 from pathlib import Path
2
3 Path('src/__pypackages__').mkdir(parents=True, exist_ok=True)
4 Path('.editorconfig').rename('src/.editorconfig')
```

Notice that the `pathlib` code puts the path first because of method chaining!

As the Zen of Python says, "namespaces are one honking great idea, let's do more of those". The `os` module is a very large namespace with a bunch of stuff in it. **The pathlib.Path class is a much smaller and more specific namespace than the os module**. Plus the methods in this `Path` namespace return `Path` objects, which allows for method chaining instead of nested string-iful function calls.

Don't forget about the glob module!

The `os` and `os.path` modules aren't the only filepath/filesystem-related utilities in the Python standard library. The `glob` module is another handy path-related module.

We can use the `glob.glob` function for finding files that match a certain pattern:

```
1 from glob import glob
2
3 top_level_csv_files = glob('*.csv')
4 all_csv_files = glob('**/*.csv', recursive=True)
```

The new `pathlib` module includes glob-like utilities as well.

```
1 from pathlib import Path
2
3 top_level_csv_files = Path.cwd().glob('*.csv')
4 all_csv_files = Path.cwd().rglob('*.csv')
```

After you've started using `pathlib` more heavily, **you can pretty much forget about the glob module entirely**: you've got all the glob functionality you need with `Path` objects.

pathlib makes the simple cases simpler

The `pathlib` module makes a number of complex cases somewhat simpler, but **it also makes some of the simple cases even simpler**.

Need to read all the text in one or more files?

You could open the file, read its contents and close the file using a `with` block:

```
from glob import glob

file_contents = []
for filename in glob('**/*.py', recursive=True):
    with open(filename) as python_file:
        file_contents.append(python_file.read())
```

Or you could use the `read_text` method on `Path` objects and a list comprehension to read the file contents into a new list all in one line:

```
from pathlib import Path

file_contents = [
    path.read_text()
    for path in Path.cwd().rglob('*.py')
]
```

What if you need to write to a file?

You could use the `open` context manager again:

```
with open('.editorconfig') as config:
    config.write('# config goes here')
```

Or you could use the `write_text` method:

```
Path('.editorconfig').write_text('# config goes here')
```

If you prefer using `open`, whether as a context manager or otherwise, you could instead use the `open` method on your `Path` object:

```
from pathlib import Path

path = Path('.editorconfig')
with path.open(mode='wt') as config:
    config.write('# config goes here')
```

Or, as of Python 3.6, you can even pass your `Path` object to the built-in `open` function:

```
from pathlib import Path

path = Path('.editorconfig')
with open(path, mode='wt') as config:
    config.write('# config goes here')
```

Path objects make your code more explicit

What do the following 3 variables point to? What do their values represent?

```
person = '{"name": "Trey Hunner", "location": "San Diego"}'
pycon_2019 = "2019-05-01"
home_directory = '/home/trey'
```

Each of those variables points to a string.

Those strings represent different things: one is a JSON blob, one is a date, and one is a file path.

These are a little bit more useful representations for these objects:

```
from datetime import date
from pathlib import Path

person = {"name": "Trey Hunner", "location": "San Diego"}
pycon_2019 = date(2019, 5, 1)
home_directory = Path('/home/trey')
```

JSON objects deserialize to dictionaries, dates are represented natively using `datetime.date` objects, and **filesystem paths can now be generically represented using `pathlib.Path` objects**.

Using `Path` objects makes your code more explicit. If you're trying to represent a date, you can use a `date` object. If you're trying to represent a filepath, you can use a `Path` object.

I'm not a strong advocate of object-oriented programming. Classes add another layer of abstraction and abstractions can sometimes add more complexity than simplicity. But the `pathlib.Path` class is **a useful abstraction**. It's also quickly becoming a universally recognized abstraction.

Thanks to PEP 519, file path objects are now becoming the standard for working with paths. As of Python 3.6, the built-in `open` function and the various functions in the `os`, `shutil`, and `os.path` modules all work properly with `pathlib.Path` objects. **You can start using pathlib today without changing most of your code that works with paths**!

What's missing from pathlib?

While `pathlib` is great, it's not all-encompassing. There are definitely **a few missing features I've stumbled upon that I wish the `pathlib` module included**.

The first gap I've noticed is the lack of `shutil` equivalents within the `pathlib.Path` methods.

While you can pass `Path` objects (and path-like objects) to the higher-level `shutil` functions for copying/deleting/moving files and directories, there's no equivalent to these functions on `Path` objects.

So to copy a file you still have to do something like this:

```
1 from pathlib import Path
2 from shutil import copyfile
3
4 source = Path('old_file.txt')
5 destination = Path('new_file.txt')
6 copyfile(source, destination)
```

There's also no `pathlib` equivalent of `os.chdir`.

This just means you'll need to import `chdir` if you ever need to change the current working directory:

```
1 from pathlib import Path
2 from os import chdir
3
4 parent = Path('..')
5 chdir(parent)
```

The `os.walk` function has no `pathlib` equivalent either. Though you can make your own `walk`-like functions using `pathlib` fairly easily.

My hope is that `pathlib.Path` objects might eventually include methods for some of these missing operations. But even with these missing features, **I still find it much more manageable to use "`pathlib` and friends" than "`os.path` and friends"**.

Should you always use pathlib?

Since Python 3.6, **pathlib.Path objects work nearly everywhere you're already using path strings**. So I see no reason *not* to use `pathlib` if you're on Python 3.6 (or higher).

If you're on an earlier version of Python 3, you can always wrap your `Path` object in a `str` call to get a string out of it when you need an escape hatch back to string land. It's awkward but it works:

```
1 from os import chdir
2 from pathlib import Path
3
4 chdir(Path('/home/trey'))  # Works on Python 3.6+
5 chdir(str(Path('/home/trey')))  # Works on earlier versions also
```

Regardless of which version of Python 3 you're on, I would recommend giving `pathlib` a try.

And if you're stuck on Python 2 still (the clock is ticking!) the third-party pathlib2 module on PyPI is a backport so you can use `pathlib` on any version of Python.

I find that using `pathlib` often makes my code more readable. Most of my code that works with files now defaults to using `pathlib` and I recommend that you do the same. **If you can use `pathlib`, you should**.

If you'd like to continue reading about pathlib, check out my follow-up article called No really, pathlib is great.

Posted by Trey Hunner Dec 21st, 2018 2:00 pm python

« Python Cyber Monday SalesNo really, pathlib is great »

Comments

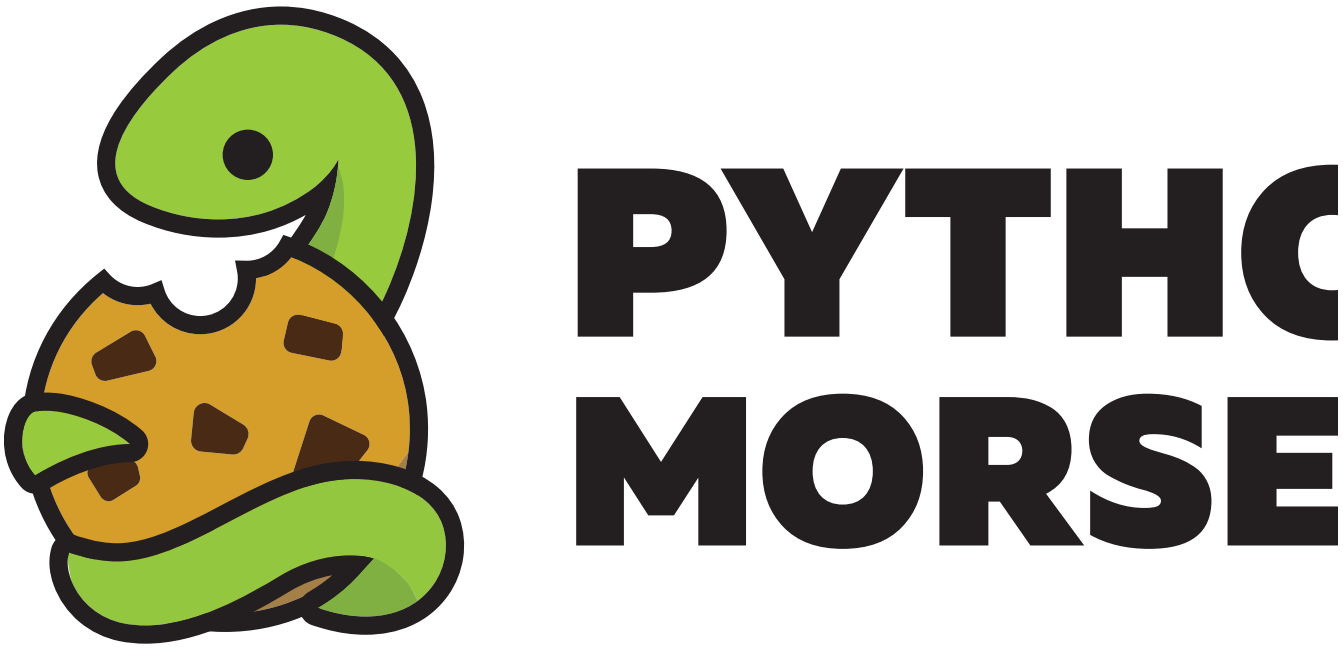
Hi! My name is Trey Hunner.

I help Python teams **write better Python code** through Python team training.

I also help individuals **level-up their Python skills** with weekly Python skill-building.

Python Team Training

Write Pythonic code



The best way to improve your skills is to **write more code**, but it's time consuming to figure out what code to write. I've made a Python skill-building service to help solve this problem.

Each week you'll get an exercise that'll help you dive deeper into Python and carefully **reflect on your own coding style**. The first 4 exercises are free.

Sign up below for **four free exercises**!

Your email Sign up

See the Python Morsels Privacy Policy.
This form is reCAPTCHA protected (see Google Privacy Policy & Terms of Service)

Favorite Posts

- Python List Comprehensions
- How to Loop With Indexes in Python
- Check Whether All Items Match a Condition in Python
- Keyword (Named) Arguments in Python: How to Use Them
- Tuple unpacking improves Python code readability
- The Idiomatic Way to Merge Dictionaries in Python
- The Iterator Protocol: How for Loops Work in Python
- Craft Your Python Like Poetry
- Python: range is not an iterator!
- Counting Things in Python: A History

Follow @treyhunner