1 FIELD GUIDE

Getting started with Pathlib

everyday superpowers

Previously, whenever any of us have needed to reference a file or a folder on our operating system, we have had to import and use the `os` and `os.path` modules. They have been great tools to get our work done, but honestly for a while, I thought I was doing something wrong.

In order to access a folder two levels up from the python file I was working in, I would have to do this:

```
import os.path

folder = os.path.dirname(os.path.dirname(os.path.abspath(__
file__)))
```

Whenever I typed this out, I thought that I was doing things wrong. This never seemed like python to me. The long namespace made me feel like I was writing too much code. The function names were confusing to what they did. Also in order to understand what this line did, I had to skip most of contents of the line, to resolve the code from the inside of the most nested parentheses out.

But since python 3.4, python has shipped with a new module in the standard library that makes working with file systems much easier and more elegant. It's called `pathlib`, and I think it can be your newest favorite module!

Using a `Path` object from the `pathlib` module, you can do things like iterate through the json files in a folder, read the contents of a file, or rename a file, just by accessing methods on the `Path` object.

It's really a nice tool to familiarize yourself with. It can make your code easier to work with, and save yourself a headache or two.

Getting started

At the top of a python file, you will need to import the one thing you need:

```
from pathlib import Path
```

That's right! You don't need to use `pip` to install it; it's a part of the standard library.

Setting a reference point

Most projects I work on need a reference point. This is the folder where from which everything else can be found. It's either the most important folder for your program or a central location that can easily access others. This reference point will depend on what your program does.

For example, Django projects use two variables that point to important folders. The `BASE_DIR` variable points to the root of your Django project, and `STATIC_ROOT` is a variable that points to where Django will place static assets.

The most common options would probably be:

- the folder your python file lives in

- the root folder of your package
- the current working folder

Let's see how to do this for each case:

Get the location of your python file

The majority of the time, you may find that you should start identifying your important location relative to a specific python file.

To set your reference point with the location of the python file you're writing in, you would do this:

```
this_file = Path(__file__)
```

If you're not familiar with `__file__`, it's a variable that python creates at runtime with the name of the python file in scope, like "example.py".

Get the folder your python file is in

While it's nice to get the location of a specific python file. It's rare that you will only want to know its location. Most of the time, you'd like to know what folder it's in.

With `Path`, there are a few ways to do that. Here are three ways to get the folder of the current python file:

```
this_folder1 = Path(__file__, '..')
this_folder2 = Path(__file__) / '..'
this_folder3 = Path(__file__).parent
```

While these three variables actually point to the same folder, they are not equal... at least not yet.

```
# This will fail:
assert this_folder1 == this_folder2 == this_folder3
```

At this point, if you were to print out the variables, they would look like this:

```
Path('example.py/..')  # this_folder1
Path('example.py/..')  # this_folder2
Path('.')  # this_folder3
```

This looks a little odd, because these are relative paths. To make them equal, they need to be absolute paths. To ensure a `Path` object is an absolute path, it needs to be resolved.

The `resolve()` method removes `'..'` segments, follows symlinks, and returns the absolute path to the item.

```
# this works
assert this_folder1.resolve() == this_folder2.resolve() == this_
folder3.resolve()
```

Get the folder two levels up

Sometimes you'll need to get the folder above the folder you're in, especially if your project's root folder is two folders up. A very good way to do this is this way:

```
# robust
two_levels_up = Path(__file__).resolve(strict=True).parents[1]
```

Note: Since python 3.6, `resolve()` takes an optional `strict` argument. If the path doesn't exist and `strict` is `True`, `FileNotFoundError` is raised. If `strict` is `False`, the path is resolved as far as possible and any remainder is appended without checking whether it exists. Before version 3.6, `resolve` always acted in a strict manner.

This is a "bulletproof" way of finding the folder, that holds the folder, that holds your python file. It'll follow symlinks and throws an error if somehow it doesn't exist.

Most of the time, you won't need something this robust. But if you're code is going to get installed somewhere else, it's good to know about.

If you're happy using something more basic, you can do this:

```
two_levels_up = Path(__file__).resolve().parents[1]
```

Warning: As of python 3.8, You need to `resolve()` your `Path` before accessing its `parents` attribute. Otherwise, python won't know the parents of a given `Path`.

The `Path().parents` attribute is a generator that contains all the parent folders of your current `Path` object. Whereas the `Path().parent` attribute returns the `Path` object representing the folder containing the current `Path` object.

The following example is run on a Unix / MacOS machine:

```
>>> path = Path('foo/ber/baz/boo/boom')
>>> path.parent
Path('foo/ber/baz/boo/')
>>> path.parents
<PosixPath.parents>
>>> list(path.parents)
[PosixPath('foo/ber/baz/boo'), PosixPath('foo/ber/baz'),
PosixPath('foo/ber'), PosixPath('foo'), PosixPath('.')]
```

Getting the current working folder

Sometimes the most important location for your program isn't relative to a python file. This is often the case when you're creating software that interacts with files or folders that are passed in from the command line.

Getting the current working folder is as easy as calling `Path.cwd()`.

You do have other options. You can also create a new `Path` object by creating an empty `Path` object:

```
folder_where_python_was_run = Path.cwd()
# Calling Path() is the same as calling Path('.')
assert Path() == Path('.')
```

The difference is that `Path.cwd()` will give you a resolved path to the current working directory, whereas creating a new `Path` object will return a relative path:

```
>>> Path.cwd()
PosixPath('/Users/chris/starting_with_pathlib')
>>> Path()
PosixPath('.')
```

You would have to `resolve()` the relative `Path` object to use it's full potential, so using `Path.cwd()` will probably be the better thing to do in most cases.

Going elsewhere

Once you have your base folder, you can build paths to anywhere you want. There are many ways you can do this.

The following shows a few examples of creating variables to a number of folders based off a variable set to the project root folder.

```
project_root = Path(__file__).resolve().parents[1]
static_files = project_root / 'static'
media_files = Path(project_root, 'media')

compiled_js_folder = static_files.joinpath('dist', 'js')
compiled_css_folder = static_files / 'dist/css'
optimized_image_folder = static_files / 'dist' / 'img'
```

Note: **All of these will work** in Unix, MacOS, and Windows! This is one of the best things about the `pathlib` module. Any path you create will work on any platform. Gone are the days you have to worry about your code working in a different operating system!

In fact, the moment you interact with a `Path` object, it's already been converted to a platform-specific object. For example, when I check the type of a `Path` on my Mac, I see it's already converted.

```
>>> type(Path())
<class 'pathlib.PosixPath'
```

Working with files

Now that you have your program's most important folders locked in, let's look at how you can find and interact with the files inside of them.

Identifying items in a folder

Sometimes, you may want to interact with every item in a folder. Below, I've created a list out of the items in the `static_files` folder:

```
>>> list(static_files.iterdir())
[PosixPath('src/static/css'),
 PosixPath('src/static/js'),
 PosixPath('src/static/node_modules'),
 PosixPath('src/static/package-lock.json'),
 PosixPath('src/static/scripts'),
 PosixPath('src/static/gulpfile.js')
]
```

Identifying folders in a folder

Other times, you may want to identify whether an item in a folder is itself a folder.

In this case, I'll create a list of the folders that are inside of the `static_files` folder:

```
>>> [x for x in static_files.iterdir if x.is_dir()]
[PosixPath('src/static/css'),
 PosixPath('src/static/js'),
 PosixPath('src/static/node_modules'),
 PosixPath('src/static/scripts'),
]
```

Identifying files in a folder

This is essentially the opposite of what we just covered; identifying if a `Path` is a file.

```
>>> [x for x in static_files.iterdir if x.is_file()]
[PosixPath('src/static/package-lock.json'),
 PosixPath('src/static/gulpfile.js')
]
```

Identifying similar files in a folder with `glob`

Sometimes you are looking for a particular kind of file in a folder. The `glob` method looks in one folder for anything that matches the pattern you pass in.

```
>>> list(compiled_js_folder.glob('*.js'))
[PosixPath('static/dist/js/app.min.js'),
 PosixPath('static/dist/js/app.js')
]
```

The `Path.glob` method can take shell-style wildcards arguments that include `*`, `?`, and `[]` characters. It acts the same as the `glob` module in the standard library. Look at the end of the glob documentation for more information.

Find similar files with `rglob`

The `rglob` method takes the same type of arguments as `glob`, but it recursively looks for matches in every folder under the path.

If you're familiar with glob patterns, it is similar to adding `**/` to the beginning of a `glob` argument.

```
>>> sorted(project_root.rglob('*.js'))
[PosixPath('static/gulpfile.js'),
 PosixPath('static/js/app.js'),
 PosixPath('static/js/app.min.js'),
 PosixPath('static/node_modules/@fullhuman/postcss-purgecss/lib/
postcss-purgecss.esm.js'),
 ...
]
```

Making sure a `Path` exists

Whenever you want to be sure the `Path` you are working with is valid, you can see if it `exists()`.

```
>>> Path('relative/path/to/nowhere').exists()
False
```

Doing things with files

Easily read text

If you want to read the entire contents of a file into memory as text, the `Path` obljject makes it easy!

```
contents = (project_root / 'read.me').read_text()
```

Read bytes

Likewise, if you want to read the entire contents of a file into memory as bytes:

```
contents = media_files.joinpath('image.jpg').read_bytes()
```

Read only part of a file at a time

Often, it would be better to not read the whole file at once. Most of the time in python, we read a file one line at a time using the `open()` built-in function.

The `Path().open()` works like the `open()` built-in!

```
with Path('sample.txt').open() as f:
    for line in f:
        process_line(line)
```

Write files

A `Path` object has similar methods for writing data to files.

If you want to write one line at a time, use the `open()` method:

```
with open(Path('example.txt'), 'w') as f:
    f.write('Sample contents')
```

If you have the contents of the file in memory, there are also convenience methods to get it to a file quickly:

```
Path('example.txt').write_text('Sample contents')
```

There is also a `Path().write_bytes()` method.

Parts of the `Path` object

In addition to these methods, a `Path` object has attributes to get at elements of the file:

```
>>> Path('static/js/app.js').name
'app.js'
>>> Path('static/js/app.js').stem
'app'
>>> Path('static/js/app.js').suffix
'.js'
```

Be careful with files with multiple extensions

Keep in mind that some files may have multiple suffixes. In this case, `Path` has the `suffixes` attribute:

```
>>> Path('static/dist/js/app.min.js').name
'app.min.js'
>>> Path('static/dist/js/app.min.js').suffix
'.js'
>>> Path('static/dist/js/app.min.js').suffixes
'.min.js'
>>> Path('static/dist/js/app.min.js').stem
'app'
```

Using `Path` with other tools

Since Python 3.4, you can use `Path()` in most places where a `PathLike` object is called for.

This includes `os.path.join()` and the built-in `open()` method.

```
# this works
sample_path = os.path.join(Path.cwd(), 'sample')

# Options that work with json
import json
```

```
data = json.load((static_files / 'package-lock.json').open())

# or
data = json.loads((static_files / 'package-lock.json').read_
text())
```

Conclusion

I hope this inspires you to try working with the `pathlib` module. I have found it very enjoyable to work with, and I hope the same for you.

More on pathlib can be found at the pathlib documentation at https://docs.python.org/3/library/pathlib.html.