Published in Towards Data Science

Christopher Tao

Feb 14, 2021 · 7 min read ★ · Listen

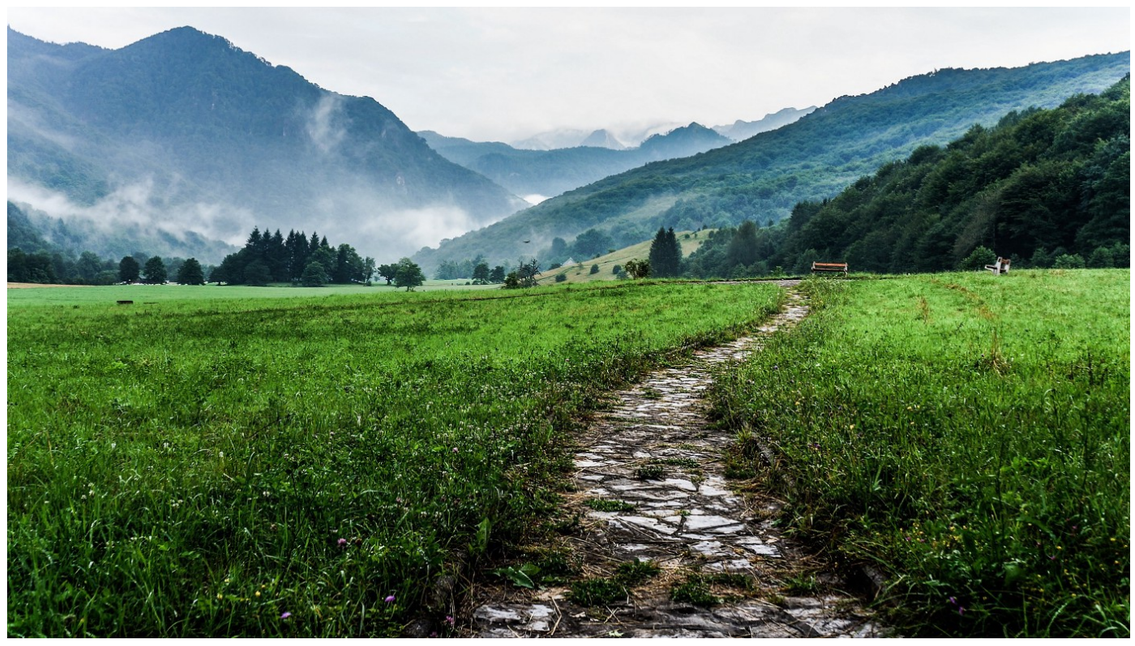

Photo by Free-Photos on Pixabay

Don’t Use Python OS Library Any More When Pathlib Can Do

More intuitive, More convenient, More features

In recent years, Python is known by a lot more people who are not programmers. This is not only because of its popularity in the area of Machine Learning but also because it could be used to automate a lot of repetitive works such as bulk editing files with certain patterns.

In one of my previous articles (as follows), I have introduced the OS library in Python which will handle almost all the basic file system operations. These operations are highly recommended for those who have just started their journey to use Python to automate some repetitive tasks.

8 Must-Know File System Operations In Python
The essential for Python in tasks automation apps
towardsdatascience.com

However, after we have mastered the OS library, I would also recommend stepping up to another library for most of the basic file system operations. That is the Pathlib, which is also a Python built-in library. It is more intuitive in terms of syntax, easier to use and has more features out of the box.

In this article, I’ll first compare some of the basic operations using the OS library and Pathlib to demonstrate the differences, as well as argue why Pathlib is recommended.

1. Show Current Directory

The first operation that I want to start with is to show the Current Working Directory (CWD). It is important because we might want to use relative paths most of the time. Therefore, it is sometimes important to know where we are at the moment.

OS Library

```
import os

os.getcwd()
```

Pathlib

```
from pathlib import Path

Path.cwd()
```

Difference

The outcome seems to be the same if we print them.

```
[2]  print(f'Current Working Directory is: {os.getcwd()}')

     Current Working Directory is: /content

[3]  print(f'Current Working Directory is: {Path.cwd()}')

     Current Working Directory is: /content
```

However, if we check the type of the objects, we’ll see that they are different.

```
print(f"Type of os.getcwd() is: {type(os.getcwd())}")
print(f"Type of Path.cwd() is: {type(Path.cwd())}")

Type of os.getcwd() is: <class 'str'>
Type of Path.cwd() is: <class 'pathlib.PosixPath'>
```

The OS library will return a string, whereas the Pathlib will return an object of PosixPath. The benefit of having PosixPath returned is that we can directly use the returned object to do a lot more further operations. This will be demonstrated in later sections.

2. Check Directory or File Existing

In my case, I’m using Google Colab and there is a folder automatically created called “sample_data” every time a new notebook is provisioned. If I want to check whether there is such a directory, the following code will do.

OS Library

The function `os.path.exists()` takes a string type argument, which can be either the name of a directory or a file.

```
os.path.exists('sample_data/README.md')
```

```
[5]  os.path.exists('sample_data/README.md')

     True

[6]  os.path.exists('foobar')

     False
```

Pathlib

When using the Pathlib, we simply pass the path as a string to the “Path” class, which will give us a “PosixPath” object. To test whether the “PosixPath” instance is existing in the file system, just call its method `exist()` , which is more intuitive than the OS library.

```
Path('sample_data/README.md').exists()
```

```
[7]  Path('sample_data/README.md').exists()

     True

[8]  Path('foobar').exists()

     False
```

Difference

In terms of this operation, there is almost no difference between the two libraries. However, we can potentially write our code in this style when using Pathlib.

```
readme = Path('sample_data/README.md')
readme.exists()
```

Although we implemented this operation using one more line of code, we have got a reference for the `readme.md` file. In other words, we can use the variable `readme` later on for any other operations without having to pass the full path as a string again.

3. Create a Directory

Now, let’s create a directory called `test_dir` in our working directory, and see what are the differences between the OS library and the Pathlib.

OS Library

It is fairly easy to create a directory in the OS library.

```
os.mkdir('test_dir')
```



```
[13] print(f"test_dir existing: {os.path.exists('test_dir')}")
     os.mkdir('test_dir')
     print(f"test_dir existing: {os.path.exists('test_dir')}")

     test_dir existing: False
     test_dir existing: True
```

Pathlib

When it comes to the Pathlib, the syntax is quite intuitive.

```
Path('test_dir').mkdir()
```

The difference in Suppressing FileExistsError

In the OS library, when we are trying to create a directory that is already existing, an error will be thrown.

```
[10] os.mkdir('test_dir')

     ---------------------------------------------------------------------------
     FileExistsError                           Traceback (most recent call last)
     <ipython-input-10-48d7b58469aa> in <module>()
     ----> 1 os.mkdir('test_dir')

     FileExistsError: [Errno 17] File exists: 'test_dir'
```

In my previous article, it has been suggested that we should always check the existence of a directory before creating it.

```
if not os.path.exists('test_dir'):
    os.mkdir('test_dir')
```

However, if we’re using Pathlib, it becomes much easier to handle this error.

```
Path('test_dir').mkdir(exist_ok=True)
```

The function `mkdir()` accepts a flag `exist_ok`. When it is set to true, the `FileExistsError` error is automatically ignored, which is equivalent to what we have done to the OS version implementation by adding an if-condition.

The difference in Creating Multi-level Depth Directory

Another major difference is for creating a directory when its parent directories are not existing. In the OS library, we have to use a different function to achieve this. Rather than `mkdir()`, we have to use `makedirs()`.

```
os.makedirs(os.path.join('test_dir', 'level_1a', 'level_2a',
'level_3a'))
```

It does the job for sure. However, we have to always remember to use a different function.

If we’re using the Pathlib, again, we just need to set the flag `parents` to true.

```
Path(os.path.join('test_dir', 'level_1b', 'level_2b',
'level_3b')).mkdir(parents=True)
```

```
[16] os.makedirs(os.path.join('test_dir', 'level_1a', 'level_2a', 'level_3a'))
```

```
[17] Path(os.path.join('test_dir', 'level_1b', 'level_2b', 'level_3b')).mkdir(parents=True)
```

Don’t forget these are all flags for the same function. In other words, we can use both `exist_ok` and `parents` flags at the same time!

4. Show Directory Content

OS Library

When we want to show the content of a directory, it is easy in the OS library.

```
os.listdir('sample_data')
```

```
[18] os.listdir('sample_data')
```

```
['README.md',
 'anscombe.json',
 'california_housing_test.csv',
 'mnist_train_small.csv',
 'mnist_test.csv',
 'california_housing_train.csv']
```

Pathlib

The syntax to show the content of a directory in Pathlib has nothing surprise as follows.

```
Path('sample_data').iterdir()
```

```
Path('sample_data').iterdir()
```

```
<generator object Path.iterdir at 0x7feb9572cca8>
```

The difference in Returned Format

If we pay attention to the returned format of the Pathlib, it is actually providing a generator rather than a list of strings. We can get everything by popular all objects into a list from this generator.

```
list(Path('sample_data').iterdir())
```

```
[19] list(Path('sample_data').iterdir())
```

```
[PosixPath('sample_data/README.md'),
 PosixPath('sample_data/anscombe.json'),
 PosixPath('sample_data/california_housing_test.csv'),
 PosixPath('sample_data/mnist_train_small.csv'),
 PosixPath('sample_data/mnist_test.csv'),
 PosixPath('sample_data/california_housing_train.csv')]
```

However, in most cases, the reason that we want to get all the files in a directory is to perform some action one by one. Therefore, a generator would be more efficient if we just want to loop them once.

The difference in Using Glob

The OS library doesn’t provide the feature to search files using wildcards. Therefore, we have to import another library called Glob to help.

```
from glob import glob
list(glob(os.path.join('sample_data', '*.csv')))
```

```
[20] from glob import glob
```

```
[21] list(glob(os.path.join('sample_data', '*.csv')))
```

```
['sample_data/california_housing_test.csv',
 'sample_data/mnist_train_small.csv',
 'sample_data/mnist_test.csv',
 'sample_data/california_housing_train.csv']
```

If we’re using the Pathlib, very luckily, it comes with the “glob” feature.

```
list(Path('sample_data').glob('*.csv'))
```

5. Quick Read/Write to File

This feature is unique to Pathlib. When we want to read or write something to a file, it is quite common to use the following approach.

```
with open("test.txt", 'rw') as file:
    print(file.read())
```

This is indeed the standard in Python. However, sometimes we might just want to write very few bytes of stuff into a file. In this case, we can do that very easily using Pathlib.

```
f = Path('test_dir/test.txt'))
f.write_text('This is a sentence.')
```

Of course, we can also quickly read the content of the file to a string variable.

```
f.read_text()
```

```
[24] f = Path('test_dir/test.txt'))
```

```
[25] f.write_text('This is a sentence.')
```

```
19
```

```
[26] f.read_text()
```

```
'This is a sentence.'
```

6. Metadata of the File

In practice, it is quite common that we need some specific information about a file. Now, let me demonstrate how Pathlib can extract information and statistics

about a file easily for us.

Let’s keep using the file that we have used in the previous section, which is the variable `f`.

```
print(f'Path of the file is:\n{f}')
```

```
[27] print(f'Path of the file is:\n{f}')

     Path of the file is:
     test_dir/test.txt
```

Although we want to use the relative path in most cases, it is sometimes still necessary to check the absolute path of a file. We can do this very easily using Pathlib.

```
f.resolve()
```

```
[28] print(f'Absolute path of the file is:\n{f.resolve()}')

     Absolute path of the file is:
     /content/test_dir/test.txt
```

Sometimes, we may want to get the file name only by getting rid of the file extension. Or, the opposite, we want to extract the extension name of the file. Both are easy by accessing the attributes of the file.

```
f.stem
f.suffix
```

```
[29] print(f'File name without extension:\n{f.stem}')

     File name without extension:
     test
```

```
[30] print(f'File extention:\n{f.suffix}')

     File extention:
     .txt
```

Even most stunning, Pathlib can return the statistics, create/update time and so on easily from a PosixPath instance. This is equivalent to the `os.stat_result`, but much easier to be accessed and consumed.

```
f.stat()
```

```
[31] print(f'File stats:\n{f.stat()}')

     File stats:
     os.stat_result(st_mode=33188, st_ino=3801354, st_dev=52, st_nlink=1, st_uid=0, st_gid=0, st_size=19, st_atime=1613299490, st_mtime=1613299490, st_ctime=1613299490)
```

For example, if we want to show the size of the file, just as follows.

```
f.stat().st_size
```

```
[32] print(f'File size (Bytes):\n{f.stat().st_size}')

     File size (Bytes):
     19
```

Summary

Photo by Hermann on Pixabay

In this article, I have introduced another Python built-in library, the Pathlib. It is considered to be more advanced, convenient and provides more stunning features than the OS library.

Of course, we still need to know how to use the OS library as it is one of the most powerful and basic libraries in Python. However, when we need some file system operations in typical scenarios, it is highly recommended to use Pathlib.

If you feel my articles are helpful, please consider joining Medium Membership to support me and thousands of other writers! (Click the link above)