**Goodbye os.path: 15 Pathlib Tricks to Quickly Master The File System in Python**

**No headaches and unreadable code from os.path**

Pathlib may be my favorite library (after Sklearn, obviously). And given there are over 130 thousand libraries, that’s saying something. Pathlib helps me turn code like this written in os.path:

import os  
  
dir\_path = "/home/user/documents"  
  
# Find all text files inside a directory  
files = [os.path.join(dir\_path, f) for f in os.listdir(dir\_path) \  
 if os.path.isfile(os.path.join(dir\_path, f)) and f.endswith(".txt")]

into this:

from pathlib import Path  
  
# Find all text files inside a directory  
files = list(dir\_path.glob("\*.txt"))

Pathlib came out in Python 3.4 as a replacement for the nightmare that was os.path. It also marked an important milestone for Python language on the whole: they finally turned every single thing into an object (even [nothing](https://docs.python.org/3/c-api/none.html)).

The biggest drawback of os.path was treating system paths as strings, which led to unreadable, messy code and a steep learning curve.

By representing paths as fully-fledged **objects**, Pathlib solves all these issues and introduces elegance, consistency, and a breath of fresh air into path handling.

And this long-overdue article of mine will outline some of the best functions/features and tricks of pathlib to perform tasks that would have been truly horrible experiences in os.path.

Learning these features of Pathlib will make everything related to paths and files easier for you as a data professional, especially during data processing workflows where you have to move around thousands of images, CSVs, or audio files.

Let’s get started!

**Working with paths**

**1. Creating paths**

Almost all features of pathlib is accessible through its Path class, which you can use to create paths to files and directories.

There are a few ways you can create paths with Path. First, there are class methods like cwd and home for the current working and the home user directories:

from pathlib import Path  
  
Path.cwd()

PosixPath('/home/bexgboost/articles/2023/4\_april/1\_pathlib')

Path.home()

PosixPath('/home/bexgboost')

You can also create paths from string paths:

p = Path("documents")  
  
p

PosixPath('documents')

Joining paths is a breeze in Pathlib with the forward slash operator:

data\_dir = Path(".") / "data"  
csv\_file = data\_dir / "file.csv"  
  
print(data\_dir)  
print(csv\_file)

data  
data/file.csv

Please, don't let anyone ever catch you using os.path.join after this.

To check whether a path, you can use the boolean function exists:

data\_dir.exists()

True

csv\_file.exists()

True

Sometimes, the entire Path object won’t be visible, and you have to check whether it is a directory or a file. So, you can use is\_dir or is\_file functions to do it:

data\_dir.is\_dir()

True

csv\_file.is\_file()

True

Most paths you work with will be relative to your current directory. But, there are cases where you have to provide the exact location of a file or a directory to make it accessible from any Python script. This is when you use absolute paths:

csv\_file.absolute()

PosixPath('/home/bexgboost/articles/2023/4\_april/1\_pathlib/data/file.csv')

Lastly, if you have the misfortune of working with libraries that still require string paths, you can call str(path):

str(Path.home())

'/home/bexgboost'

Most libraries in the data stack have long supported Path objects, including sklearn, pandas, matplotlib, seaborn, etc.

**2. Path attributes**

Path objects have many useful attributes. Let’s see some examples using this path object that points to an image file.

image\_file = Path("images/midjourney.png").absolute()  
  
image\_file

PosixPath('/home/bexgboost/articles/2023/4\_april/1\_pathlib/images/midjourney.png')

Let's start with the parent. It returns a path object that is one level up the current working directory.

image\_file.parent

PosixPath('/home/bexgboost/articles/2023/4\_april/1\_pathlib/images')

Sometimes, you may want only the file name instead of the whole path. There is an attribute for that:

image\_file.name

'midjourney.png'

which returns only the file name with the extension.

There is also stem for the file name without the suffix:

image\_file.stem

'midjourney'

Or the suffix itself with the dot for the file extension:

image\_file.suffix

'.png'

If you want to divide a path into its components, you can use parts instead of str.split('/'):

image\_file.parts

('/',  
 'home',  
 'bexgboost',  
 'articles',  
 '2023',  
 '4\_april',  
 '1\_pathlib',  
 'images',  
 'midjourney.png')

If you want those components to be Path objects in themselves, you can use parents attribute, which creates a generator:

for i in image\_file.parents:  
 print(i)

/home/bexgboost/articles/2023/4\_april/1\_pathlib/images  
/home/bexgboost/articles/2023/4\_april/1\_pathlib  
/home/bexgboost/articles/2023/4\_april  
/home/bexgboost/articles/2023  
/home/bexgboost/articles  
/home/bexgboost  
/home  
/

**Working with files**

To create files and write to them, you don't have to use open function anymore. Just create a Path object and write\_text or write\_btyes to them:

markdown = data\_dir / "file.md"  
  
# Create (override) and write text  
markdown.write\_text("# This is a test markdown")

Or, if you already have a file, you can read\_text or read\_bytes:

markdown.read\_text()

'# This is a test markdown'

len(image\_file.read\_bytes())

1962148

However, note that write\_text or write\_bytes overrides existing contents of a file.

# Write new text to existing file  
markdown.write\_text("## This is a new line")

# The file is overridden  
markdown.read\_text()

'## This is a new line'

To append new information to existing files, you should use open method of Path objects in a (append) mode:

# Append text  
with markdown.open(mode="a") as file:  
 file.write("\n### This is the second line")  
  
markdown.read\_text()

'## This is a new line\n### This is the second line'

It is also common to rename files. rename method accepts the destination path for the renamed file.

To create the destination path in the current directory, i. e. rename the file, you can use with\_stem on the existing path, which replaces the stem of the original file:

renamed\_md = markdown.with\_stem("new\_markdown")  
  
markdown.rename(renamed\_md)

PosixPath('data/new\_markdown.md')

Above, file.md is turned into new\_markdown.md.

Let's see the file size through stat().st\_size:

# Display file size  
renamed\_md.stat().st\_size

49 # in bytes

or the last time the file was modified, which was a few seconds ago:

from datetime import datetime  
  
modified\_timestamp = renamed\_md.stat().st\_mtime  
  
datetime.fromtimestamp(modified\_timestamp)

datetime.datetime(2023, 4, 3, 13, 32, 45, 542693)

st\_mtime returns a timestamp, which is the count of seconds since January 1, 1970. To make it readable, you can use use the fromtimestamp function of datatime.

To remove unwanted files, you can unlink them:

renamed\_md.unlink(missing\_ok=True)

Setting missing\_ok to True won't raise any alarms if the file doesn't exist.

**Working with directories**

There are a few neat tricks to work with directories in Pathlib. First, let's see how to create directories recursively.

new\_dir = (  
 Path.cwd()  
 / "new\_dir"  
 / "child\_dir"  
 / "grandchild\_dir"  
)  
  
new\_dir.exists()

False

The new\_dir doesn't exist, so let's create it with all its children:

new\_dir.mkdir(parents=True, exist\_ok=True)

By default, mkdir creates the last child of the given path. If the intermediate parents don't exist, you have to set parents to True.

To remove empty directories, you can use rmdir. If the given path object is nested, only the last child directory is deleted:

# Removes the last child directory  
new\_dir.rmdir()

To list the contents of a directory like ls on the terminal, you can use iterdir. Again, the result will be a generator object, yielding directory contents as separate path objects one at a time:

for p in Path.home().iterdir():  
 print(p)

/home/bexgboost/.python\_history  
/home/bexgboost/word\_counter.py  
/home/bexgboost/.azure  
/home/bexgboost/.npm  
/home/bexgboost/.nv  
/home/bexgboost/.julia  
...

To capture all files with a specific extension or a name pattern, you can use the glob function with a regular expression.

For example, below, we will find all text files inside my home directory with glob("\*.txt"):

home = Path.home()  
text\_files = list(home.glob("\*.txt"))  
  
len(text\_files)

3 # Only three

To search for text files recursively, meaning inside all child directories as well, you can use *recursive glob* with rglob:

all\_text\_files = [p for p in home.rglob("\*.txt")]  
  
len(all\_text\_files)

5116 # Now much more

Learn about regular expressions [here](https://realpython.com/regex-python/).

You can also use rglob('\*') to list directory contents recursively. It is like the supercharged version of iterdir().

One of the use cases of this is counting the number of file formats that appear within a directory.

To do this, we import the Counter class from collections and provide all file suffixes to it within the articles folder of home:

from collections import Counter  
  
file\_counts = Counter(  
 path.suffix for path in (home / "articles").rglob("\*")  
)  
  
file\_counts

Counter({'.py': 12,  
 '': 1293,  
 '.md': 1,  
 '.txt': 7,  
 '.ipynb': 222,  
 '.png': 90,  
 '.mp4': 39})

**Operating system differences**

Sorry, but we have to talk about this nightmare of an issue.

Up until now, we have been dealing with PosixPath objects, which are the default for UNIX-like systems:

type(Path.home())

pathlib.PosixPath

If you were on Windows, you would get a WindowsPath object:

from pathlib import WindowsPath  
  
# User raw strings that start with r to write windows paths  
path = WindowsPath(r"C:\users")  
path

NotImplementedError: cannot instantiate 'WindowsPath' on your system

Instantiating another system's path raises an error like the above.

But what if you were forced to work with paths from another system, like code written by coworkers who use Windows?

As a solution, pathlib offers pure path objects like PureWindowsPath or PurePosixPath:

from pathlib import PurePosixPath, PureWindowsPath  
  
path = PureWindowsPath(r"C:\users")  
path

PureWindowsPath('C:/users')

These are primitive path objects. You've access to some path methods and attributes, but essentially, the path object remains a string:

path / "bexgboost"

PureWindowsPath('C:/users/bexgboost')

path.parent

PureWindowsPath('C:/')

path.stem

'users'

path.rename(r"C:\losers") # Unsupported

AttributeError: 'PureWindowsPath' object has no attribute 'rename'

**Conclusion**

If you have noticed, I lied in the title of the article. Instead of 15, I believe the count of new tricks and functions was 30ish.

I didn't want to scare you off.

But I hope I've convinced you enough to ditch os.path and start using pathlib for much easier and more readable path operations.

Forge a new *path*, if you will :)