Have you struggled with file path handling in Python? In Python 3.4 and above, the struggle is now over! You no longer need to scratch your head over code like:

```
Python >>>
>>> path.rsplit('\\', maxsplit=1)[0]
```

Or cringe at the verbosity of:

```
Python >>>
>>> os.path.isfile(os.path.join(os.path.expanduser('~'), 'realpython.txt'))
```

In this tutorial, you will see how to work with file paths—names of directories and files—in Python. You will learn new ways to read and write files, manipulate paths and the underlying file system, as well as see some examples of how to list files and iterate over them. Using the `pathlib` module, the two examples above can be rewritten using elegant, readable, and Pythonic code like:

```
Python >>>
>>> path.parent
>>> (pathlib.Path.home() / 'realpython.txt').is_file()
```

Free PDF Download: Python 3 Cheat Sheet

The Problem With Python File Path Handling

Working with files and interacting with the file system are important for many different reasons. The simplest cases may involve only reading or writing files, but sometimes more complex tasks are at hand. Maybe you need to list all files in a directory of a given type, find the parent directory of a given file, or create a unique file name that does not already exist.

Traditionally, Python has represented file paths using regular text strings. With support from the `os.path` standard library, this has been adequate although a bit cumbersome (as the second example in the introduction shows). However, since paths are not strings, important functionality is spread all around the standard library, including libraries like `os`, `glob`, and `shutil`. The following example needs three `import` statements just to move all text files to an archive directory:

```
Python
import glob
import os
import shutil

for file_name in glob.glob('*.txt'):
    new_path = os.path.join('archive', file_name)
    shutil.move(file_name, new_path)
```

With paths represented by strings, it is possible, but usually a bad idea, to use regular string methods. For instance, instead of joining two paths with `+` like regular strings, you should use `os.path.join()`, which joins paths using the correct path separator on the operating system. Recall that Windows uses `\` while Mac and Linux use `/` as a separator. This difference can lead to hard-to-spot errors, such as our first example in the introduction working for only Windows paths.

The `pathlib` module was introduced in Python 3.4 (PEP 428) to deal with these challenges. It gathers the necessary functionality in one place and makes it available through methods and properties on an easy-to-use `Path` object.

Early on, other packages still used strings for file paths, but as of Python 3.6, the `pathlib` module is supported throughout the standard library, partly due to the addition of a file system path protocol. If you are stuck on legacy Python, there is also a backport available for Python 2.

Time for action: let us see how `pathlib` works in practice.

Creating Paths

All you really need to know about is the `pathlib.Path` class. There are a few different ways of creating a path. First of all, there are classmethods like `.cwd()` (Current Working Directory) and `.home()` (your user's home directory):

Python >>>

```
>>> import pathlib
>>> pathlib.Path.cwd()
PosixPath('/home/gahjelle/realpython/')
```

> **Note:** Throughout this tutorial, we will assume that `pathlib` has been imported, without spelling out `import pathlib` as above. As you will mainly be using the `Path` class, you can also do `from pathlib import Path` and write `Path` instead of `pathlib.Path`.

A path can also be explicitly created from its string representation:

Python >>>

```
>>> pathlib.Path(r'C:\Users\gahjelle\realpython\file.txt')
WindowsPath('C:/Users/gahjelle/realpython/file.txt')
```

A little tip for dealing with Windows paths: on Windows, the path separator is a backslash, `\`. However, in many contexts, backslash is also used as an *escape character* in order to represent non-printable characters. To avoid problems, use *raw string literals* to represent Windows paths. These are string literals that have an `r` prepended to them. In raw string literals the `\` represents a literal backslash: `r'C:\Users'`.

A third way to construct a path is to join the parts of the path using the special operator `/`. The forward slash operator is used independently of the actual path separator on the platform:

Python >>>

```
>>> pathlib.Path.home() / 'python' / 'scripts' / 'test.py'
PosixPath('/home/gahjelle/python/scripts/test.py')
```

The `/` can join several paths or a mix of paths and strings (as above) as long as there is at least one `Path` object. If you do not like the special `/` notation, you can do the same thing with the `.joinpath()` method:

Python >>>

```
>>> pathlib.Path.home().joinpath('python', 'scripts', 'test.py')
PosixPath('/home/gahjelle/python/scripts/test.py')
```

Note that in the preceding examples, the `pathlib.Path` is represented by either a `WindowsPath` or a `PosixPath`. The actual object representing the path depends on the underlying operating system. (That is, the `WindowsPath` example was run on Windows, while the `PosixPath` examples have been run on Mac or Linux.) See the section Operating

System Differences for more information.

Reading and Writing Files

Traditionally, the way to read or write a file in Python has been to use the built-in `open()` function. This is still true as the `open()` function can use `Path` objects directly. The following example finds all headers in a Markdown file and prints them:

Python

```
path = pathlib.Path.cwd() / 'test.md'
with open(path, mode='r') as fid:
    headers = [line.strip() for line in fid if line.startswith('#')]
print('\n'.join(headers))
```

An equivalent alternative is to call `.open()` on the `Path` object:

Python

```
with path.open(mode='r') as fid:
    ...
```

In fact, `Path.open()` is calling the built-in `open()` behind the scenes. Which option you use is mainly a matter of taste.

For simple reading and writing of files, there are a couple of convenience methods in the `pathlib` library:

- `.read_text()`: open the path in text mode and return the contents as a string.
- `.read_bytes()`: open the path in binary/bytes mode and return the contents as a bytestring.
- `.write_text()`: open the path and write string data to it.
- `.write_bytes()`: open the path in binary/bytes mode and write data to it.

Each of these methods handles the opening and closing of the file, making them trivial to use, for instance:

Python >>>

```
>>> path = pathlib.Path.cwd() / 'test.md'
>>> path.read_text()
<the contents of the test.md-file>
```

Paths can also be specified as simple file names, in which case they are interpreted relative to the current working directory. The following example is equivalent to the previous one:

Python >>>

```
>>> pathlib.Path('test.md').read_text()
<the contents of the test.md-file>
```

The `.resolve()` method will find the full path. Below, we confirm that the current working directory is used for simple file names:

```
Python >>>
>>> path = pathlib.Path('test.md')
>>> path.resolve()
PosixPath('/home/gahjelle/realpython/test.md')

>>> path.resolve().parent == pathlib.Path.cwd()
True

>>> path.parent == pathlib.Path.cwd()
False
```

Note that when paths are compared, it is their representations that are compared. In the example above, `path.parent` is not equal to `pathlib.Path.cwd()`, because `path.parent` is represented by `'.'` while `pathlib.Path.cwd()` is represented by `'/home/gahjelle/realpython/'`.

Picking Out Components of a Path

The different parts of a path are conveniently available as properties. Basic examples include:

- `.name`: the file name without any directory
- `.parent`: the directory containing the file, or the parent directory if path is a directory
- `.stem`: the file name without the suffix
- `.suffix`: the file extension
- `.anchor`: the part of the path before the directories

Here are these properties in action:

```
Python >>>
>>> path
PosixPath('/home/gahjelle/realpython/test.md')
>>> path.name
'test.md'
>>> path.stem
'test'
>>> path.suffix
'.md'
>>> path.parent
PosixPath('/home/gahjelle/realpython')
>>> path.parent.parent
PosixPath('/home/gahjelle')
>>> path.anchor
'/'
```

Note that `.parent` returns a new `Path` object, whereas the other properties return strings. This means for instance that `.parent` can be chained as in the last example or even combined with `/` to create completely new paths:

```
Python >>>
>>> path.parent.parent / ('new' + path.suffix)
PosixPath('/home/gahjelle/new.md')
```

The excellent Pathlib Cheatsheet provides a visual representation of these and other properties and methods.

Moving and Deleting Files

Through `pathlib`, you also have access to basic file system level operations like moving, updating, and even deleting files. For the most part, these methods do not give a warning or wait for confirmation before information or files are lost. Be careful when using these methods.

To move a file, use `.replace()`. Note that if the destination already exists, `.replace()` will overwrite it. Unfortunately, `pathlib` does not explicitly support safe moving of files. To avoid possibly overwriting the destination path, the simplest is to test whether the destination exists before replacing:

Python

```
if not destination.exists():
    source.replace(destination)
```

However, this does leave the door open for a possible race condition. Another process may add a file at the `destination` path between the execution of the `if` statement and the `.replace()` method. If that is a concern, a safer way is to open the destination path for exclusive creation and explicitly copy the source data:

Python

```
with destination.open(mode='xb') as fid:
    fid.write(source.read_bytes())
```

The code above will raise a `FileExistsError` if `destination` already exists. Technically, this copies a file. To perform a move, simply delete `source` after the copy is done (see below). Make sure no exception was raised though.

When you are renaming files, useful methods might be `.with_name()` and `.with_suffix()`. They both return the original path but with the name or the suffix replaced, respectively.

For instance:

Python >>>

```
>>> path
PosixPath('/home/gahjelle/realpython/test001.txt')
>>> path.with_suffix('.py')
PosixPath('/home/gahjelle/realpython/test001.py')
>>> path.replace(path.with_suffix('.py'))
```

Directories and files can be deleted using `.rmdir()` and `.unlink()` respectively. (Again, be careful!)

Examples

In this section, you will see some examples of how to use `pathlib` to deal with simple challenges.

Counting Files

There are a few different ways to list many files. The simplest is the `.iterdir()` method, which iterates over all files in the given directory. The following example combines `.iterdir()` with the `collections.Counter` class to count how many files there are of each filetype in the current directory:

Python >>>

```
>>> import collections
>>> collections.Counter(p.suffix for p in pathlib.Path.cwd().iterdir())
Counter({'.md': 2, '.txt': 4, '.pdf': 2, '.py': 1})
```

More flexible file listings can be created with the methods `.glob()` and `.rglob()` (recursive glob). For instance, `pathlib.Path.cwd().glob('*.txt')` returns all files with a `.txt` suffix in the current directory. The following only counts filetypes starting with `p`:

Python >>>

```
>>> import collections
>>> collections.Counter(p.suffix for p in pathlib.Path.cwd().glob('*.p*'))
Counter({'.pdf': 2, '.py': 1})
```

Display a Directory Tree

The next example defines a function, `tree()`, that will print a visual tree representing the file hierarchy, rooted at a given directory. Here, we want to list subdirectories as well, so we use the `.rglob()` method:

Python

```
def tree(directory):
    print(f'+ {directory}')
    for path in sorted(directory.rglob('*')):
        depth = len(path.relative_to(directory).parts)
        spacer = '    ' * depth
        print(f'{spacer}+ {path.name}')
```

Note that we need to know how far away from the root directory a file is located. To do this, we first use `.relative_to()` to represent a path relative to the root directory. Then, we count the number of directories (using the `.parts` property) in the representation. When run, this function creates a visual tree like the following:

Python >>>

```
>>> tree(pathlib.Path.cwd())
+ /home/gahjelle/realpython
    + directory_1
        + file_a.md
    + directory_2
        + file_a.md
        + file_b.pdf
        + file_c.py
    + file_1.txt
    + file_2.txt
```

> **Note:** The f-strings only work in Python 3.6 and later. In older Pythons, the expression `f'{spacer}+ {path.name}'` can be written `'{0}+ {1}'.format(spacer, path.name)`.

Find the Last Modified File

The `.iterdir()`, `.glob()`, and `.rglob()` methods are great fits for generator expressions and list comprehensions. To find the file in a directory that was last modified, you can use the `.stat()` method to get information about the underlying files. For instance, `.stat().st_mtime` gives the time of last modification of a file:

```
Python >>>
>>> from datetime import datetime
>>> time, file_path = max((f.stat().st_mtime, f) for f in directory.iterdir())
>>> print(datetime.fromtimestamp(time), file_path)
2018-03-23 19:23:56.977817 /home/gahjelle/realpython/test001.txt
```

You can even get the contents of the file that was last modified with a similar expression:

```
Python >>>
>>> max((f.stat().st_mtime, f) for f in directory.iterdir())[1].read_text()
<the contents of the last modified file in directory>
```

The timestamp returned from the different `.stat().st_` properties represents seconds since January 1st, 1970. In addition to `datetime.fromtimestamp`, `time.localtime` or `time.ctime` may be used to convert the timestamp to something more usable.

Create a Unique File Name

The last example will show how to construct a unique numbered file name based on a template. First, specify a pattern for the file name, with room for a counter. Then, check the existence of the file path created by joining a directory and the file name (with a value for the counter). If it already exists, increase the counter and try again:

```
Python
def unique_path(directory, name_pattern):
    counter = 0
    while True:
        counter += 1
        path = directory / name_pattern.format(counter)
        if not path.exists():
            return path

path = unique_path(pathlib.Path.cwd(), 'test{:03d}.txt')
```

If the directory already contains the files `test001.txt` and `test002.txt`, the above code will set `path` to `test003.txt`.

Operating System Differences

Earlier, we noted that when we instantiated `pathlib.Path`, either a `WindowsPath` or a `PosixPath` object was returned. The kind of object will depend on the operating system you are using. This feature makes it fairly easy to write cross-platform compatible code. It is possible to ask for a `WindowsPath` or a `PosixPath` explicitly, but you will only be limiting your code to that system without any benefits. A concrete path like this can not be used on a different system:

```
Python >>>
>>> pathlib.WindowsPath('test.md')
NotImplementedError: cannot instantiate 'WindowsPath' on your system
```

There might be times when you need a representation of a path without access to the underlying file system (in which case it could also make sense to represent a Windows path on a non-Windows system or vice versa). This can be done with `PurePath` objects. These objects support the operations discussed in the section on Path Components but not the methods that access the file system:

```
Python >>>
>>> path = pathlib.PureWindowsPath(r'C:\Users\gahjelle\realpython\file.txt')
>>> path.name
'file.txt'
>>> path.parent
PureWindowsPath('C:/Users/gahjelle/realpython')
>>> path.exists()
AttributeError: 'PureWindowsPath' object has no attribute 'exists'
```

You can directly instantiate `PureWindowsPath` or `PurePosixPath` on all systems. Instantiating `PurePath` will return one of these objects depending on the operating system you are using.

Paths as Proper Objects

In the introduction, we briefly noted that paths are not strings, and one motivation behind `pathlib` is to represent the file system with proper objects. In fact, the official documentation of `pathlib` is titled *`pathlib` — Object-oriented filesystem paths*. The Object-oriented approach is already quite visible in the examples above (especially if you contrast it with the old `os.path` way of doing things). However, let me leave you with a few other tidbits.

Independently of the operating system you are using, paths are represented in Posix style, with the forward slash as the path separator. On Windows, you will see something like this:

```
Python >>>
>>> pathlib.Path(r'C:\Users\gahjelle\realpython\file.txt')
WindowsPath('C:/Users/gahjelle/realpython/file.txt')
```

Still, when a path is converted to a string, it will use the native form, for instance with backslashes on Windows:

```
Python >>>
>>> str(pathlib.Path(r'C:\Users\gahjelle\realpython\file.txt'))
'C:\\Users\\gahjelle\\realpython\\file.txt'
```

This is particularly useful if you are using a library that does not know how to deal with `pathlib.Path` objects. This is a bigger problem on Python versions before 3.6. For instance, in Python 3.5, the `configparser` standard library can only use string paths to read files. The way to handle such cases is to do the conversion to a string explicitly:

```
Python >>>
>>> from configparser import ConfigParser
>>> path = pathlib.Path('config.txt')
>>> cfg = ConfigParser()
>>> cfg.read(path)                    # Error on Python < 3.6
TypeError: 'PosixPath' object is not iterable
>>> cfg.read(str(path))               # Works on Python >= 3.4
['config.txt']
```

In Python 3.6 and later it is recommended to use `os.fspath()` instead of `str()` if you need to do an explicit conversion. This is a little safer as it will raise an error if you accidently try to convert an object that is not pathlike.

Possibly the most unusual part of the `pathlib` library is the use of the `/` operator. For a little peek under the hood, let us see how that is implemented. This is an example of operator overloading: the behavior of an operator is changed depending on the context. You have seen this before. Think about how `+` means different things for strings and numbers. Python implements operator overloading through the use of *double underscore* methods (a.k.a. *dunder*

methods).

The `/` operator is defined by the `.__truediv__()` method. In fact, if you take a look at the source code of `pathlib`, you'll see something like:

```
Python
class PurePath(object):

    def __truediv__(self, key):
        return self._make_child((key,))
```

Conclusion

Since Python 3.4, `pathlib` has been available in the standard library. With `pathlib`, file paths can be represented by proper `Path` objects instead of plain strings as before. These objects make code dealing with file paths:

- Easier to read, especially because `/` is used to join paths together
- More powerful, with most necessary methods and properties available directly on the object
- More consistent across operating systems, as peculiarities of the different systems are hidden by the `Path` object

In this tutorial, you have seen how to create `Path` objects, read and write files, manipulate paths and the underlying file system, as well as some examples of how to iterate over many file paths.