Day 35: Pathlib Library

- Python's `pathlib` module helps streamline your work with file and directory paths. Instead of relying on traditional string-based path handling, you can use the `Path` object, which provides a cross-platform way to read, write, move, and delete files.
- `pathlib` also brings together functionality previously spread across other libraries like `os`, `glob`, and `shutil`, making file operations more straightforward. Plus, it includes built-in methods for reading and writing text or binary files, ensuring a clean and Pythonic approach to handling file tasks.

The Problem With Representing Paths as Strings

- Traditionally, Python has represented file paths using regular text strings. However, since paths are more than plain strings, important functionality was spread all around the standard library, including in libraries like `os`, `glob`, and `shutil`.

  ```
  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)
  ```

  You need 3 import statements in order to move all the text files to an archive directory.

- Python's `pathlib` provides a `Path` class that works the same way on different operating systems. Instead of importing different modules such as `glob`, `os`, and `shutil`, you can perform the same tasks by using `pathlib` alone.

  ```
  from pathlib import Path
  ```

```
for file_path in Path.cwd().glob("*.txt"):
    new_path = Path("archive") / file_path.name
    file_path.replace(new_path)
```

With `pathlib`, you accomplish these tasks with fewer `import` statements and more straightforward syntax. Less imports mean more save on memory.

Path Instantiation with Python's pathlib

- Heart of `pathlib` is the `Path` class.
- Here object-oriented approach is quite visible since we focus on files and directories rather than path as string.

Using Path Methods

- Get the current working directory

  ```
  from pathlib import Path
  Path.cwd()

  # Output
  # WindowsPath('C:/Users/rohit/Desktop/realpython')
  # PosixPath('/home/rohit/Desktop/realpython')
  ```

  When you instantiate `pathlib.Path`, you get either a `WindowsPath` or a `PosixPath` object. The kind of object will depend on which operating system you're using.

  WindowsPath: Windows

  PosixPath: Linux / MacOS

- Generally, it's a good idea to use `Path`. With `Path`, you instantiate a **concrete path** for the platform that you're using while also keeping your code platform-independent. Concrete paths allow you to do system calls on path objects, but **pure paths** only allow you to manipulate paths without accessing the operating system.

- Working with platform-independent paths means that you can write a script on Windows that uses `Path.cwd()`, and it'll work correctly when you run the file on macOS or Linux. The same is true for `.home()`.

  ```
  from pathlib import Path
  Path.home()
  # Output
  # WindowsPath('C:/Users/philipp')
  ```

Passing in a String

- Instead of starting in your user's home directory or your current working directory, you can point to a directory or file directly by passing its string representation into `Path`. This process creates a `Path` object. Instead of having to deal with a string, you can now work with the flexibility that `pathlib` offers.

```
from pathlib import Path
Path(r"C:\Users\philipp\realpython\file.txt")

# WindowsPath('C:/Users/rohit/Desktop/realpython/file.txt')
```

Joining Paths

```
from pathlib import Path

for file_path in Path.cwd().glob("*.txt"):
    new_path = Path("archive") / file_path.name
    file_path.rename(new_path)
```

If you don't like the special slash notation, then you can do the same operation with the `.joinpath()` method:

```
from pathlib import Path
Path.home().joinpath("python", "scripts", "test.py")
```

```
# PosixPath('/home/gahjelle/python/scripts/test.py')
```

File System Operations With Paths

Picking out components of a Path

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

```
from pathlib import Path
path = Path(r"C:\Users\rohit\realpython\test.md")
path
# WindowsPath('C:/Users/rohit/realpython/test.md')

path.name
# 'test.md'

path.stem
# 'test'

path.suffix
# '.md'

path.anchor
# 'C:\\'

path.parent
# WindowsPath('C:/Users/rohit/realpython")
```

```
path.parent.parent
# WindowsPath('C:/Users/rohit')
```

Note that `.parent` returns a new `Path` object, whereas the other properties return strings. This means, for instance, that you can chain `.parent` in the last example or even combine it with the slash operator to create completely new path

```
path.parent.parent / f"new{path.suffix}"
# PosixPath('/home/rohit/new.md')
```

Reading and Writing Files

```
from pathlib import Path

path = Path.cwd() /"shopping_list.md"
with path.open(mode="r", encoding="utf-8") as md_file:
    content = md_file.read()
    groceries = [line for line in content.splitlines() if line.startswith("*")]
print("\n".join(groceries))
```

- Traditionally, the way to read or write a file in Python has been to use the built-in `open()` function, with additional read_text() and write_text() functions.