File Paths

When you access a file on an operating system, a file path is required. The file path is a string that represents the location of a file. It's broken up into three major parts:

1. **Folder Path:** the file folder location on the file system where subsequent folders are separated by a forward slash `/` (Unix) or backslash `\` (Windows)
2. **File Name:** the actual name of the file
3. **Extension:** the end of the file path pre-pended with a period (`.`) used to indicate the file type

Here's a quick example. Let's say you have a file located within a file structure like this:

```
/
│
├── path/
│   │
│   ├── to/
│   │   └── cats.gif
│   │
│   └── dog_breeds.txt
│
└── animals.csv
```

Let's say you wanted to access the `cats.gif` file, and your current location was in the same folder as `path`. In order to access the file, you need to go through the `path` folder and then the `to` folder, finally arriving at the `cats.gif` file. The Folder Path is `path/to/`. The File Name is `cats`. The File Extension is `.gif`. So the full path is `path/to/cats.gif`.

Now let's say that your current location or current working directory (cwd) is in the `to` folder of our example folder structure. Instead of referring to the `cats.gif` by the full path of `path/to/cats.gif`, the file can be simply referenced by the file name and extension `cats.gif`.

```
/
│
├── path/
│   │
│   ├── to/  ← Your current working directory (cwd) is here
│   │   └── cats.gif  ← Accessing this file
│   │
│   └── dog_breeds.txt
│
└── animals.csv
```

But what about `dog_breeds.txt`? How would you access that without using the full path? You can use the special characters double-dot (`..`) to move one directory up. This means that `../dog_breeds.txt` will reference the `dog_breeds.txt` file from the directory of `to`:

```
/
│
├── path/  ← Referencing this parent folder
│   │
│   ├── to/  ← Current working directory (cwd)
│   │   └── cats.gif
│   │
│   └── dog_breeds.txt  ← Accessing this file
│
└── animals.csv
```

The double-dot (`..`) can be chained together to traverse multiple directories above the current directory. For example, to access `animals.csv` from the `to` folder, you would use `../../animals.csv`.

Line Endings

One problem often encountered when working with file data is the representation of a new line or line ending. The line ending has its roots from back in the Morse Code era, when a specific pro-sign was used to communicate the end of a transmission or the end of a line.

Later, this was standardized for teleprinters by both the International Organization for Standardization (ISO) and the American Standards Association (ASA). ASA standard states that line endings should use the sequence of the Carriage Return (`CR` or `\r`) *and* the Line Feed (`LF` or `\n`) characters (`CR+LF` or `\r\n`). The ISO standard however allowed for either the `CR+LF` characters or just the `LF` character.

Windows uses the `CR+LF` characters to indicate a new line, while Unix and the newer Mac versions use just the `LF` character. This can cause some complications when you're processing files on an operating system that is different than the file's source. Here's a quick example. Let's say that we examine the file `dog_breeds.txt` that was created on a Windows system:

```
Pug\r\n
Jack Russell Terrier\r\n
English Springer Spaniel\r\n
German Shepherd\r\n
Staffordshire Bull Terrier\r\n
Cavalier King Charles Spaniel\r\n
Golden Retriever\r\n
West Highland White Terrier\r\n
Boxer\r\n
Border Terrier\r\n
```

This same output will be interpreted on a Unix device differently:

```
Pug\r
\n
Jack Russell Terrier\r
\n
English Springer Spaniel\r
\n
German Shepherd\r
\n
Staffordshire Bull Terrier\r
```

```
\n
Cavalier King Charles Spaniel\r
\n
Golden Retriever\r
\n
West Highland White Terrier\r
\n
Boxer\r
\n
Border Terrier\r
\n
```

This can make iterating over each line problematic, and you may need to account for situations like this.

Reading and Writing Opened Files

Once you've opened up a file, you'll want to read or write to the file. First off, let's cover reading a file. There are multiple methods that can be called on a file object to help you out:

Method	What It Does
`.read(size=-1)`	This reads from the file based on the number of `size` bytes. If no argument is passed or `None` or `-1` is passed, then the entire file is read.
`.readline(size=-1)`	This reads at most `size` number of characters from the line. This continues to the end of the line and then wraps back around. If no argument is passed or `None` or `-1` is passed, then the entire line (or rest of the line) is read.
`.readlines()`	This reads the remaining lines from the file object and returns them as a list.