File reading/writing

Examples

Python is able to read and modify text files (and binary files, too) using the `open` function. In the example below, a file 'file.txt' was previously created on disk.

In [1]:

```python
with open("file.txt") as f:
    line_1 = f.readline()
    line_2 = f.readline()

print(line_1, line_2, sep="")
```

```
A new line
Another new line
```

By default, `open` opens the file:

- in **text mode**
- in **read only** mode: modifying the file is not possible

```
In [2]:
with open("file.txt") as f:
    new_line = f.write("A new line\nAnother new line\n")
```

```
---------------------------------------------------------------------------
UnsupportedOperation                      Traceback (most recent call last)
Cell In[2], line 2
      1 with open("file.txt") as f:
----> 2     new_line = f.write("A new line\nAnother new line\n")

UnsupportedOperation: not writable
```

To edit the file, one can use one of the following options:

- `'w'`: write to the beginning of the file and existing content is removed!
- `'a'`: write at the end of the file, existing content is kept

In [3]:

```python
with open("file.txt", "a") as f:
    new_line = f.write("A new line\nAnother new line\n")

with open("file.txt") as f:
    print("Appending to existing file: ", end="")
    print(f.readlines())

with open("file.txt", "w") as f:
    new_line = f.write("A new line\nAnother new line\n")

with open("file.txt") as f:
    print("Replacing content of existing file: ", end="")
    print(f.readlines())
```

```
Appending to existing file: ['A new line\n', 'Another new line\n', 'A new li
ne\n', 'Another new line\n']
Replacing content of existing file: ['A new line\n', 'Another new line\n']
```

Note the following methods:

- `readline` : read a single line of a file. If several calls to `readline` are done, lines are displayed one after another.
- `readlines` : read all the lines of the file and store them into a list
- `write` : write a string in the file

Notes

`\n` is the universal character to describe a line break:

In [4]:

```
s = "This is a sentence.\nA"
print(s[-3:])   # the last three caracters are 'A',
                # a new line and a dot '.':
                # the new line is not made of 2 caracters!
```

```
.
A
```

Loading [MathJax]/extensions/Safe.js

The `with` bloc is important: it makes sure file is open and closed in a clean way.

The other way to manage files is described here after: it is **depreciated** because if `f.close()` is never called then the file might be corrupted or damage the operating system.

In [5]:

```
f = open("file.txt")
line_1 = f.readline()
line_2 = f.readline()
f.close()   # never forget this one!
print(line_1, line_2, sep="")
```

```
A new line
Another new line
```

One can create an empty text file:

In [6]:

```
with open("my_empty_file.txt", "w") as f:
    pass
```

Loading [MathJax]/extensions/Safe.js

Files management

Key ideas

A file path is the adress of a file on the disk. In Python, the preferred way to handle file paths is to use the **`pathlib`** library (built in). It handles perfectly the differences of separators ('/' or '') between different operating systems. `pathlib.Path` instances can handle both files **and** directories.

In [7]:

```python
from pathlib import Path
path = Path("/this/is/my/path/a_file.txt")
print(path.name)    # file
print(path.parent)  # directory
print(path.suffix)  # file extension
```

```
a_file.txt
/this/is/my/path
.txt
```

The creation of a `Path` instance does not mean the corresponding path exists:

```
In [8]: path.exists()
```

```
Out[8]: False
```

Yet, it can be used to create it:

In [9]:
```
if False:
    path.parent.mkdir(
                        parents=True,    # if parents do not exist, they are created
                        exist_ok=True    # if the path already exists, it does nothin
                        )
```

Loading [MathJax]/extensions/Safe.js

Absolute and relative file paths

An absolute path is a complete adress of a file (or directory) on the disk. Using Linux, these paths start with '/' (root), using Windows they start with the drive name ('c:/', 'd:/', etc...).

```
In [10]: path
```

```
Out[10]: PosixPath('/this/is/my/path/a_file.txt')
```

```
In [11]: path.is_absolute()
```

```
Out[11]: True
```

In [12]:

```python
relative_path = Path("path/relative/to/current/directory")
print(relative_path.is_absolute())
```

```
False
```

Conversely, relative paths are path defined starting from the current directory, which can be obtained using `Path.cwd()`. This directory is also called `'.'`. Th e parent of this directory is called '`..`'.

In [13]:

```python
path1 = Path("./dir1/dir2/dir3/../..")
path2 = Path("./dir1/")
print(path1)
print(path2)
```

```
dir1/dir2/dir3/../..
dir1
```

Two relative paths cannot be compared. One must first call the `resolve` method that returns an absolute path.

In [14]:

```python
print(path1==path2)
print(path1.resolve()==path2.resolve())
```

```
False
True
```

Some libraries do not accept `Path` instances...in this case one must use `str`.

In [15]:

```python
if False:
    print(str(path1.resolve()))
```

Define complex paths

The `/` operator creates a single path from two paths. It can be used several times in a row:

In [16]:

```
base_path = Path("/my/project/is/in/a/very/deep/dir")
data_path = base_path / "data" / "case_study"
src_path = data_path / "../../src"
file_path = data_path / "a_file.txt"
print(base_path.resolve())
print(data_path.resolve())
print(src_path.resolve())
print(file_path.resolve())
```

```
/my/project/is/in/a/very/deep/dir
/my/project/is/in/a/very/deep/dir/data/case_study
/my/project/is/in/a/very/deep/dir/src
/my/project/is/in/a/very/deep/dir/data/case_study/a_file.txt
```

Loading [MathJax]/extensions/Safe.js

Browse your files

Let's create a fictive files structure:

```
A/
    1/
        a/
            file_1.txt
            file_2.txt
        b/
            file_1.txt
            file_2.txt
    2/
        a/
            file_1.txt
            file_2.txt
        b/
            file_1.txt
            file_2.txt
    useless_file.txt
```

A list of the files of a specific directory is available using the `iterdir` method of a `Path` instance describing this directory.

`iterdir` returns a generator. Below, it is transformed into a list for easier handling:

In [17]:

```python
p = Path('A')
print(p.iterdir())
print(list(p.iterdir()))
```

```
<generator object Path.iterdir at 0x71888b55de50>
[PosixPath('A/useless_file.txt'), PosixPath('A/useless_file_new_name.txt'),
PosixPath('A/2'), PosixPath('A/1')]
```

One can also browse sub directories using the `walk` function of library **`os`** (built in).

In [18]:

```python
import os
for current_dir, subdirs, files in os.walk(p):
    print(f"{current_dir:<8}", subdirs, files)
```

```
A        ['2', '1'] ['useless_file.txt', 'useless_file_new_name.txt']
A/2      ['b', 'a'] []
A/2/b    [] ['file_1', 'file_2']
A/2/a    [] ['file_1', 'file_2']
A/1      ['b', 'a'] []
A/1/b    [] ['file_1', 'file_2']
A/1/a    [] ['file_1', 'file_2']
```

Note that:

- `os.walk` returns strings
- a method `Path.walk` exists for very recent version of python, and should be prefered over `os.walk` if available

Operations on files

Removal

A file can be removed using `Path.unlink`. if it's a directory, then use `Path.rmdir`.

In [19]:

```
if False:
    p = Path('A/useless_file.txt')
    print("Existing files: ", list(p.parent.iterdir()))
    p.unlink()
    print("A file was removed: ", list(p.parent.iterdir()))
```

Move/copy

To move or copy/paste a file, the `shutil` library must be used (built in).

Below, a file is moved to the same directory, but its name is changed:

In [20]:
```python
import shutil
source = Path('A/useless_file.txt')
destination = Path('A/useless_file_new_name.txt')

shutil.move(source, destination)
```

Out[20]:
```
PosixPath('A/useless_file_new_name.txt')
```

Copy/paste:

In [21]:

```
shutil.move(destination, source)
source = Path('A/useless_file.txt')
destination = Path('A/useless_file_new_name.txt')

shutil.copy2(source, destination)   #  source file still exists
```

Out[21]:

```
PosixPath('A/useless_file_new_name.txt')
```

Note: the copy of metadata (owner of the file, permissions, dates, etc...) might fail!

Take away

3 libraries can handle files:

- browse the disk, delete files and directories: use `pathlib` (documentation) in priority, else `os` (documentation).
- move, copy files and directories: use `shutil` (documentation)

Run a system call

Introduction

The call to an external program from Python makes it possible to build complex scripts that involve several different software components.

The key idea is to define a command the same way one would define it in a terminal (Linux, OS X) or a *cmd* command line (Windows).

Example

One must use the `run` function of library `subprocess` (built in).

In [22]:

```python
import subprocess
name = 'something'
result = subprocess.run(args=f"mkdir {name}",
                        shell=True,
                        capture_output=True,
                        check=True)
print(type(result))
print(result)
```

Loading [MathJax]/extensions/Safe.js

```
---------------------------------------------------------------------------
CalledProcessError                        Traceback (most recent call last)
Cell In[22], line 3
      1 import subprocess
      2 name = 'something'
----> 3 result = subprocess.run(args=f"mkdir {name}",
      4                         shell=True,
      5                         capture_output=True,
      6                         check=True)
      7 print(type(result))
      8 print(result)

File /usr/lib/python3.12/subprocess.py:571, in run(input, capture_output, ti
meout, check, *popenargs, **kwargs)
    569     retcode = process.poll()
    570     if check and retcode:
--> 571         raise CalledProcessError(retcode, process.args,
    572                                  output=stdout, stderr=stderr)
    573 return CompletedProcess(process.args, retcode, stdout, stderr)

CalledProcessError: Command 'mkdir something' returned non-zero exit status
1.
```

Some explanations:

- `args` describes the command to run
- `shell=True` allows to specify `args` as a `str`, if `shell=False`, then `args` must be set to `['mkdir', 'a_new_dir']`
- `capture_output=True` stores the outputs of the command in the attributes `stdout` (standard output) and `stderr` (error output) of the instance returned by `run` (here this instance is `results)
- `check=True` makes sure an error is raised if the system command (described by `args`) fails
- attribute `returncode` of `results` is 0 when the command **succeeds**

Loading [MathJax]/extensions/Safe.js