I'm not robot

You should seek to the start of the file before modifying it using `file.truncate()` for in-place replacement. Here's how you can do it: ```python import re myfile = "path/test.xml" with open(myfile, "r+") as f: data = f.read() f.seek(0) f.write(re.sub(r"ABC(\s+)(.*)", r"ABC\1\2", data)) f.truncate() ``` Alternatively, you can read the file, then open it again in write mode: ```python with open(myfile, "r") as f: data = f.read() with open(myfile, "w") as f: f.write(re.sub(r"ABC(\s+)(.*)", r"ABC\1\2", data)) ``` Note that neither `truncate()` nor opening the file in write mode will change its inode number. This is not specific to Python, but rather how the underlying file system works. You can also use the `python-magic` library to determine the MIME type of a file without calling an external shell command. For example: ```python import magic print(magic.from_file('iceland.jpg')) # Output: 'JPEG image data, JFIF standard 1.01' print(magic.from_file('greenland.png', mime=True)) # Output: 'image/png' ```` This library uses the same underlying `libmagic` library as the *NIX `file` command. Here's an example of how to find files recursively using Python: ````python import os from os import scandir def is_sym_link(path): return os.path.isdir(path) and (ctypes.windll.kernel32.GetFileAttributesW(str(path)) & 0x0400) def find(base, filenames): hits = [] def find_in_dir_subdir(direc): content = scandir(direc) for entry in content: if entry.name in filenames: hits.append(os.path.join(direc, entry.name)) elif entry.is_dir() and not is_sym_link(os.path.join(direc, entry.name)): try: find_in_dir_subdir(os.path.join(direc, entry.name)) except UnicodeDecodeError: print("Could not resolve " + os.path.join(direc, entry.name)) except PermissionError: print("Skipped " + os.path.join(direc, entry.name) + ". I lacked permission to navigate") continue if not os.path.exists(base): return [] else: find_in_dir_subdir(base) return hits # Example usage: print(find("C:\\", ["Python", "Homework"])) ``` Note that this code uses the `scandir` function from the `os` module, which returns an iterator over the directory entries. It also uses the `ctypes` library to call the Windows API function `GetFileAttributesW`. I finally managed to get it working after making some adjustments. All credit goes to @F.M.F. You can indeed display a file open dialog using tkinter in Python 2 or Tkinter in Python 3 (see other answers for details). However, keep in mind that the UI of this dialog is outdated and doesn't match newer Windows 10 file open dialogs. Also, if you're looking to embed Python support into your own app, you'll find out that tkinter is not an open-source library and has a commercial pricing model (e.g., search for "activetcl pricing"). I found the pythonnet library instead, which is open-source (MIT License). To install it, use pip3: pip3 install pythonnet. Here's an example of using the file open dialog with pythonnet: import sys import ctypes import clr clr.AddReference('System.Windows.Forms') from System.Windows.Forms import OpenFileDialog file_dialog = OpenFileDialog() ret = file_dialog.ShowDialog() if ret != 1: print("Cancelled") sys.exit() print(file_dialog.FileName) If you need a more complex UI, check the Demo folder in pythonnet's git. I'm not sure about portability to other OS's, but .net 5 is planned to be multi-OS compatible. One reason why using with open('filename.txt') as fp: for line in fp: print(line) is preferred is that it ensures file handles are closed quickly enough, avoiding "too many files open" errors in hypothetical Python implementations without deterministic reference-counting garbage collection. The with block is a safer way to handle this. Bonus question: why isn't the file closing feature included in the iterator protocol for file objects? This feels wrong because it combines two separate tasks—iterating over lines and closing the file handle—in one action, which can be surprising and harder to understand for humans reading the code. The more challenging aspect of Haskell's IO system lies in its capacity for logical reasoning about program behavior. Other languages have arrived at similar conclusions. Haskell experimented with "lazy IO" which enables automatic file closure once you've reached the end of the stream, but this approach is now generally discouraged and users have largely shifted to explicit resource management solutions like Conduit, mirroring Python's with block. From a technical standpoint, certain operations involving file handles in Python wouldn't be feasible if iteration closed the file handle. For instance, consider iterating over a file twice: `with open('filename.txt') as fp: for line in fp: ... fp.seek(0) for line in fp: ...`. This scenario may arise when adding new code to an existing codebase that previously utilized similar logic. If iteration were to close the file, this wouldn't be possible. Separating iteration and resource management simplifies composing code chunks into a functioning program, a crucial usability feature of languages or APIs.

Python file name convention. Python file write. Python file extension. Python file type. Python fileinput. Python file handling. Python file size. Python filename without extension. Python file exists. Python file read. Python filenotfounderror. Python filename from path. Python file object. Python file readline. Python file path.