בס"ד

**SafeFile:**

This exercise will work on many things you have learned, including OOP, exception handling, custom exceptions, slots, file handling, the os library, packages, variable parameters to methods, and more. The exercise is split into two stages and a bonus, and each stage is split into parts. **Do not read ahead** – read only the part you are at for now, but be sure to read the entire part before starting to implement it (the build of each part is the problem requirement, and afterwards pointers to help you along).

* Time requirements – each section has the estimated amount of time it should take. Less time of course is welcome. However, if you find yourself needing a lot more time than written, it might mean that you did not interpret the exercise’s meaning correctly (of course, sometimes while programming there are unforeseen bugs and problems, this is perfectly normal and okay if it occurs).

Setup:

* Before you start, create a GIT repository for your project. Work with GIT throughout this entire exercise, including versions.
* Before you start, take a look at PEP8 <https://www.python.org/dev/peps/pep-0008/> These are the conventions used for Python. Your program should follow these standards (Conventions are for example that module names should use lower case letters and \_ to separate words, and classes should start with upper case letters and use camel case to separate words.). In Python, because all variables are dynamic, the conventions become much more important, because you rely on the way the name is written to be able to know what kind of information it contains. Especially look at these sections (PEP8 is long): <https://www.python.org/dev/peps/pep-0008/#programming-recommendations>, <https://www.python.org/dev/peps/pep-0008/#comments>, <https://www.python.org/dev/peps/pep-0008/#naming-conventions>.

Again, because this is long, limit the reading of PEP8 to no more than 15 minutes.

Stage 1

Implement the SafeFile class. SafeFile is a class that offers a limited functionality wrapper for the built in file object (it only implements read and write), but raises indicative errors for problems.

Part 1: (2 hours) Create a class that wraps an inner file object. Your class supports the following methods: a read method that receives an optional parameter for the mode, a write method that receives the content to write and an optional parameter for the mode. Your construct should receive the path of the file. (You need to be able to support the modes: r, w, a, rb, wb, ab.).

For example, your class should support the following usages:

handle = SafeFile(r'c:\file.txt')

content = handle.read()

binary\_content = handle.read(‘rb’)

handle.write("Hello world", 'a')

handle.write("Hello world")

A few pointers:

* Your class needs to support variable parameters: the read method should work with no parameters, and with one parameter indicating the mode. Read about how to do this if you are not sure.
* Your class will receive the mode parameter, and needs to pass it on to the builtin file method. Think about what happens if the user gives you a parameter that does not fit.
* Your class should use the builtin file object. Some notes on the file object:
  + A file object, once opened, is opened for a specific mode: r, w, a, and others. Read about the different modes in the docs, or by typing in IPython “file?”. Once a file is opened for a specific usage, the same handle cannot be used for another usage. For example, consider the following code:
    - file\_path = r'c:\temp\some\_file.txt'
    - message = 'Hello World'
    - with open(file\_path, 'r') as inner\_file: # Open the file for read access
    - content = inner\_file.read()
    - inner\_file.write(message) # This will fail, file is not opened for writing
    - You need to think about how you solve this problem. Hint: you can’t open the file in the \_\_init\_\_ method of the class, because you don’t know what kind of usage the user wants, and your requirement is for the same SafeFile object to allow both reading and writing.
  + Be careful to close the file object when you are finished with it. Builtin files are opened with the open() method, and must be closed later with the close() method. (The ‘with’ statement shown above is a shortcut Python offers for this). Read more about opening and closing handles in the attached document “A little bit about handles”.

Part 2: (2 hours) One of the most important things when writing a program is to have indication of what went wrong. The program's way of indicating the problems is by raising exceptions. Create a new exception class, SafeFileException. This exception indicates that something went wrong with the SafeFile usage. For any exception the builtin file object raises, catch that exception and raise SafeFileException instead, with the same message the inner exception had. SafeFileException should only be raised for errors in the file I/O, and not for any potential error – only if the builtin file raises an exception, raise this exception.

A few pointers:

* When you catch exceptions, you want to be sure of what kind of exceptions you are catching. In Python, you have the “except” clause, which can catch specific types or all exceptions raised. Whenever handling exceptions, you want to be sure exactly what kind of exceptions you are handling. Read about the file object, and make sure that your except clauses catch only the relevant exceptions you want to handle.
* Notice the requirement to raise a new exception with the same message the original exception had. Read about how to do get this information from the exception object raised. (If you really want to go all the way, also print the type of the original exception raised)
* A recommendation: relevant for all the code, but keep in mind - if you have a chunk of code that repeats itself in different places, maybe it is time to separate it to a different method or a different object. (There is no requirement that the SafeFile object needs to hold a builtin file object directly...)

Part 3: (2 hours) If you read the "A little bit about handles", you should remember the 'with' statement. If you did not do so already, refactor your code to use the ‘with’ statement in all places (instead of open/close directly). Read about slots, and how the with statement is implemented. If this is relevant for your code (if you created another wrapper for the file object), refactor it to be a context manager. If not, create as a side project a small context manager (hint: read about how to implement a context manager, and slots in general)

A few pointers:

* Keep in mind that in the Python implementation of files, open is a global method that returns a file, and close is a class method of file object. In your implementation, open should be a class method. This is something you should think about - you can't just copy the open/close scheme as it appears in the builtin implementation.

Part 4: (2 hours) SafeFile needs to be safe, which means that it has to do more than just wrap the generic file object. Change the SafeFile class to run more verifications, as appears below. Create several more types of exceptions, that should ALL inherit from SafeFileException: FileContentException, FileNameException, FileNotFoundException. Each exception should have an appropriate error message that should be to the point but indicate what went wrong.

These are your verifications:

* If the content of the file contains the word 'red' or the word 'dangerous', raise FileContentException
* If the file name contains multiple extensions (file.txt.txt, or file.abc.txt), raise FileNameException
* If the file name contains no extension (no\_extension, hello\_txt), raise FileNameException
* If the file is in a directory that does not exist, FileNotFoundException

A few pointers:

* Some verifications you can do before the operation on the file. For example, you don't have to open the file to check whether its name is valid.
* Some of these exceptions the builtin file object raises for itself, and some it does not. In the cases it does, you have two options: validate and then check, or run the method and if it raises - raise the appropriate error. Read about what is considered better in Python, and in your case specifically. Also, make sure you write a comment explaining why you use the solution you chose.
* Look at the documentation for the os module, it can help you a lot. Also, the split method. In general, for many small operations you might want to do, there is a large chance an implementation exists in a library module, which means you do not have to create one yourself. Get used to searching to see whether the operation you want to do has been written before.

Part 5: (2 hours) Add more verification to your SafeFile object. Now, in addition to checking the words 'red' and 'dangerous', also the words 'fire', 'bad', and 'exception' should raise FileContentException. Also, upgrade your SafeFile to check for these words in the file name, and if the file contains these words raise FileNameException.

A few pointers:

* Like before, watch out if your code contains sections of code that repeat itself. Also, long sections of if/elif/elif/else are something that needs to be cleaned up. Think about how you prevent your code from containing these.
* When adding a lot of additional logic, keep in mind that each method should have one purpose only. This means that if in your 'read' method there is mostly code for validating the file, maybe that code should at least be separated to another method (and maybe somewhere else altogether).

---------------------------------------------------------------------------------------------

Stage 2:

Until now, we used the SafeFile class. Now we will create a program that uses it.

Part 1: (2 hours) I hope you did this before now, but if not - create a package called safe\_file (or any other indicative name you like). This module should contain the SafeFile object, and any additional classes and functions you might have created.

Read “A little about paths”. This means that from the appropriate path, the following code should run:

import safe\_file

safe\_file.SafeFile # Should not raise NameError

A few pointers:

* Read about how to create a package in Python, creating an \_\_init\_\_.py file and what it should contain.

Part 2: (1 hour) Install your package as a Python package. (Read about how to install a package first of course). Your target - to make your package accessible from any Python code on your computer, no matter which path it is running from. After completing this stage, opening IPython and typing the code from the previous section should work as well.

Part 3: (3 hours) Create a program that uses the SafeFile class. The program's UI will be the console, and it will run in a loop, accepting files from the user and displaying appropriate information. Its run loop will look like this:

Enter file: \_\_\_\_\_\_

File Content is:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# The file content appears here

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\* Done! \*\*\*\*

Do you want to empty the file (y/n): \_\_\_

\*\*\*\* Done! \*\*\*\*

Enter file: \_\_\_\_\_

If the user wants to empty the file, add a line printing

File was successfully emptied!

For any exception occurring throughout the run loop, print:

Sorry, an error occurred: "<the message>", instead of <the message> print the error that was raised. After printing the error, return to the first line: Enter file: \_\_\_\_\_

If you are given a file that does not exist, indicate to the user.

Your program needs to handle file paths in only full path format (C:\some\_dir1\some\_dir2\file.txt). Any other input should raise errors.

A few pointers:

* Make sure you handle cases where the user input is wrong. Test your program with input in another language, empty inputs, 'yn' instead of 'y' or 'n', 'nn', very long inputs. (Any additional end cases are welcome). Handle means that the program should not crash, and should have indicative error messages when necessary.
* Think about what happens if the program has an error in the middle of an operation. For example, if the user entered bad input when asked if they want to empty the file - is the file emptied? Notify the user - they probably want to know what happened. Also, what if there is an exception while running some part - in what state is the file left?
* Think about when the file is opened for writing, and when for reading
* Keep in mind program readability - elegant code, split into functions and objects as necessary. Notice that a method that is extremely long is usually a problem - it should usually be split into several smaller, accurate methods.
* Your program has to be able to handle slashes. Also, test your program for correct output in the following input builds (use any file and directory names you want):

C:\some\_dir\some\_file.txt

some\_file.txt

\some\_dir\some\_file.txt

Part 4: (3 hours) Upgrade your program to support the following features:

* The SafeFile class offers different types of exceptions. Print a different error message according to the type of exception raised, as following:
  + FileContentException --> Problem occurred with file content "<message>"
  + FileNameException --> Problem occurred with file name "<message>"
  + FileNotFoundException --> File was not found "<message>"
* Add handling of input like 'some\_file.txt'. Files without a full path should be created in a directory called C:\safe\_file\. Keep in mind you will have to check that this directory exists, and create it if it doesn't. Think about where you want this feature - you have the option of adding it to your main program, or to the safe\_file module. Notice that if you open a file with no directory it will create the file in your current working directory (as explained in "A little bit about PATHs").

The os library should be very helpful...Also check out os.path module.

* Add option to create a new file if it does not exist (in the previous section you raised an error for this case), and add indication of the file you are handling. The program flow should be like this:

Enter file: \_\_\_\_\_\_

File <path> opened successfully!

File Content is:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# The file content appears here

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\* Done! \*\*\*\*

Do you want to empty the file (y/n): \_\_\_

\*\*\*\* Done! \*\*\*\*

Enter file: \_\_\_\_

OR:

Enter file: \_\_\_\_\_\_

File <path> does not exist, create it (y/n): \_\_

File <path> created!

Enter content: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Content saved!

File Content is:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# The file content appears here

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\* Done! \*\*\*\*

Do you want to empty the file (y/n): \_\_\_

\*\*\*\* Done! \*\*\*\*

Enter file: \_\_\_\_\_

For <path> substitute the FULL PATH of the file you are handling. The previous section should have shown you that this is a valuable addition.

**BONUS**:

Part 1: (2 hours) Add a feature to your SafeFile class: block access to its inner file object (in essence, implement a private member). Do this by using a metaclass. Of course, read about metaclasses and what they enable you to do, and how they can be used to implement this feature.

Part 2: (1 hour) Add logs to your program. Logs are indications from the program on its state (feel free to read about logs online...). For example, imagine you have a professional server, and one day it stops responding. You would want to know what happened, and why. Python offers the logging library for logs. Think about the design of the logs - should they be in the main program? In the safe\_file library? Both?