# Python for File and Exception Handling

# Interview Questions

## (Practice Project)

PW SKILLS

## Easy

**1. What are the different ways to open a file in Python, and what do the various modes ('r', 'w', 'a', etc.) signify?**

**Answer: Python has multiple ways to open a file:**

- **'r' (Read Mode):** This allows you to read the file. Python raises a FileNotFoundError if the file is not found.
- **'w' (Write Mode):** Allows writing to the file. The content of the file is truncated (erased) if it already exists; if not, a new file is generated.
- 'a' (Append Mode) option allows you to write to the file without truncating it if it already exists. The file's end has data attached to it.
- **Binary mode ('b'):** This key opens files in binary mode, which is required for non-text files like executables and pictures. It can be used in conjunction with 'r', 'w', or 'a'.
- **'+' (Read/Write Mode):** Enables writing and reading concurrently. For instance, 'w+' truncates the file before permitting reading and writing, but 'r+' permits both without doing so.

**2. How does Python ensure that a file is properly closed, and what happens if you forget to close a file?**

**Answer:** Python uses the 'close()' method to make sure a file is closed correctly. But using a 'with' statement, which immediately shuts the file after the code block is executed, is a more dependable and advised method:

```
with open('file.txt', 'r') as file:
     content = file.read() # File is automatically closed here
```

Resource leaks, in which file handles are not returned to the operating system, might occur when you fail to close a file. Eventually, this may use up all of the file handles, which would result in problems when attempting to open new files.

**3. Explain how the try-except block works in Python. What is the purpose of each part?**

**Answer:** Python uses a try-except block to manage exceptions:

- **'try':** This block contains the code that could throw an exception. The control instantly shifts to the unless block in the event of an exception.
- **'except':** The exception is caught and handled by this block. To capture certain errors, you can provide the kind of exception; alternatively, you can leave it empty to catch all exceptions.

**Example:**

```
try:
    file = open('file.txt', 'r')
    content = file.read()
except FileNotFoundError:
    print("File not found.")
finally:
    file.close()
```

**4. What are the different methods to read data from a file in Python, and how do they differ?**

**Answer: Python offers multiple ways to read information from a file:**

- **read():** Reads a file's whole contents as a string.
- **readline():** Retrieves a single line as a string from the file.
- **readlines():** Reads every line from the file and outputs a list of strings for each line.

**Example:**

```
with open('file.txt', 'r') as file:
    all_content = file.read()     # Reads entire content
    first_line = file.readline()  # Reads the first line
    all_lines = file.readlines()  # Reads all lines into a list
```

**5. How does Python handle multiple except clauses? Can you give an example where multiple exceptions are caught in a single try-except block?**

**Answer:** Python lets you define numerous 'except' clauses to handle multiple exceptions in a single 'try-except' block. Different types of exceptions can be handled by different 'except' clauses.

**Example:**

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ValueError:
    print("That's not a valid number.")
except ZeroDivisionError:
    print("You can't divide by zero.")
```

When the user tries to divide by zero, 'ZeroDivisionError' is raised, and 'ValueError' is raised if the input is not an integer.

**6. What is the difference between 'r+', 'w+', and 'a+' modes in Python file handling?**

**Answer:**

- **'r+', or Read/Write Mode**, allows the file to be opened for both writing and reading. The file is not truncated; instead, the file pointer is inserted at the beginning. Python raises a FileNotFoundError if the file is not found.
- **Write/Read Mode:** 'w+': This opens the file for writing as well as reading. If the file already exists, it is created; if not, it is truncated (erased).
- **'a+' (Append/Read Mode):** Enables both writing and reading of the file. The file is not truncated; instead, the file pointer is appended to the end of the file. The file is generated if it doesn't already exist.

## Medium

**7. In Python, how can you write binary data to a file? How does writing to a text file differ from writing to a binary file?**

**Answer: The 'wb' mode in Python is used to write binary data to a file:**

```
with open('binary_file.bin', 'wb') as file:
    file.write(b'\x00\x01\x02')
```

While writing to a binary file works directly with bytes, writing to a text file requires encoding text into a particular format (such as UTF-8). Because of this, binary files can contain executable files or other non-text data like graphics.

**8. Discuss the various methods available on Python file objects, such as tell(), seek(), and truncate(). Provide use cases for each.**

**Answer:**

- **tell():** Gives back the file pointer's current location. helpful for keeping track of where you are in a file, particularly while writing or reading in sections.
- The function seek(offset, whence) shifts the file pointer to a given location. 'whence' establishes the reference point ('0' for the start, '1' for the current location, and '2' for the end of the file), while 'offset' indicates the number of bytes to transfer. helpful for moving the cursor to a certain spot so that it can be read or written from.
- **truncate(size=None):** This resizes the file to the specified dimensions. The file is truncated at the current file pointer location if the size is not given. helpful for shrinking a file's size or removing some of its information.

**9. Explain how Python handles CSV files using the csv module. How can you read a CSV file into a dictionary format, and what are the common pitfalls when dealing with CSV files?**

**Answer:** CSV file handling capabilities is available in Python using the 'csv' module:

**Reading a CSV into a dictionary:**

```
import csv

with open('file.csv', 'r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row)  # Each row is an OrderedDict
```

Common pitfalls include dealing with inconsistent delimiters, handling quoted fields that contain the delimiter, and ensuring proper encoding.

**10. What is the role of the else clause in exception handling, and how does it differ from the finally clause?**

**Answer:** When no exception is raised in the 'try' block of a 'try-except' block, the 'else' clause is executed. It is helpful for code that must execute only in the event that the 'try' block is successful.

In contrast, the 'finally' phrase is carried out whether or not an exception was requested. Usually, it's employed for cleanup tasks like resource releases or file closures.

**Example:**

```
try:
    result = 10 / int(input("Enter a number: "))
except ZeroDivisionError:
    print("You can't divide by zero.")
else:
    print("Division successful:", result)
finally:
    print("Execution complete.")
```

**11. How does Python handle JSON files? Explain the process of reading a JSON file into a Python object and writing a Python object to a JSON file.**

**Answer:** Python handles JSON files using the 'json' package. 'json.load()' may be used to read a JSON file into a Python object (such as a dictionary) and 'json.dump()' can be used to write a Python object to a JSON file.

**Example:**

```
import json

# Reading JSON
with open('file.json', 'r') as file:
    data = json.load(file)

# Writing JSON
with open('file.json', 'w') as file:
    json.dump(data, file, indent=4)
```

**12. Describe a scenario where using the 'x' file mode (exclusive creation) is advantageous. What will happen if the file already exists?**

**Answer:** To start a new file and allow writing, use the 'x' mode. Python raises a 'FileExistsError' if the file already exists. When you want to make sure that you are creating a new file and not inadvertently overwriting an old one, this mode is helpful.

**Example:**

```
try:
    with open('new_file.txt', 'x') as file:
        file.write("This is a new file.")
except FileExistsError:
    print("File already exists.")
```

**13. How does the finally clause ensure the cleanup of resources, and what are the implications of using finally with a return statement in the try block?**

**Answer:** The 'finally' clause guarantees that, regardless of what occurs in the 'try' or 'except' blocks, a certain cleaning function (such as locking or closing a file) be executed. The 'finally' block will still run before the function exits, even if the 'try' block contains a return statement.

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return "Cannot divide by zero."
    finally:
        print("Cleanup actions, if any.")

result = divide(10, 0)  # Will print the finally message before
returning
```

**14. How would you append data to an existing file without overwriting its contents, and what is the impact of using the with statement in this context?**

**Answer: 'a' mode opens an existing file so that data can be appended to it:**

```
with open('file.txt', 'a') as file:
    file.write("Appended data.\n")
```

Even in the event of a writing error, the file will always be correctly closed when the 'with' statement is used.

**15. In Python, how would you handle large files that do not fit into memory? Which methods and techniques would you use?**

**Answer:** You can use a loop to traverse over lines or use the'read(size)' method to read the file in parts when it is too big to fit in memory. By using this method, the complete file is not loaded into memory.

**Example:**

```
with open('large_file.txt', 'r') as file:
    for line in file:
        process(line)  # Process each line
```

As an alternative, you can employ memory-mapped file access (which enables random access to big files without fully loading them into memory) by using the 'mmap' module.

**16. How would you design an exception handling mechanism for a function that interacts with external resources like files or databases to ensure robustness and maintainability?**

**Answer:** The following should be taken into account while creating an exception handling system for a function that interacts with external resources:

**Specific Exceptions:** Handle particular exceptions pertaining to the external resource (e.g., 'ConnectionError' for databases, 'FileNotfoundError' for files).

**Resource Cleanup:** 'finally' is used in resource cleanup to make sure that resources such as file handles and database connections are closed or released correctly.

**Logging:** To aid in maintainability and debugging, log exceptions.
Re-raising Exceptions: After logging an exception or enclosing it in a custom exception class, there are situations in which it makes sense to raise it again.

**Example:**

```
def read_file(filename):
    try:
        with open(filename, 'r') as file:
            return file.read()
    except FileNotFoundError as e:
        log_error(e)
        raise  # Re-raise the exception after logging
    except Exception as e:
        log_error(e)
        raise CustomFileError("An error occurred while reading
the file.")
    finally:
        cleanup_actions()
```

## Hard

**17. Suppose you need to read a CSV file with varying delimiters and malformed entries. How would you preprocess and handle this in Python to ensure clean data extraction?**

**Answer: You can use Python's 'csv' module with custom languages and error handling to handle a CSV file with different delimiters and incorrect entries:**

```
import csv

with open('file.csv', 'r') as file:
    reader = csv.reader(file, delimiter=';')
    for row in reader:
        try:
            clean_row = preprocess(row)
            # Process clean_row
        except SomeError as e:
            handle_error(e)
```

**Custom Dialect:** Define a custom dialect to handle varying delimiters.
**Error Handling:** Use a try-except block within the loop to handle malformed rows.
**Preprocessing:** Write a preprocessing function to clean or standardize rows before processing.

**18. Discuss how Python's exception hierarchy works. Provide an example of a scenario where catching exceptions in a specific order is crucial, and explain why.**

**Answer:** More specialised exceptions in Python inherit from more generic ones according to the structure of the exception hierarchy. As an example, "FileNotFoundError" is a subclass of "OSError," which descends from "Exception."

Because catching a more generic exception first will prevent the more specialised exception from being caught, it is imperative to catch exceptions in a certain order. This may result in inadvertent actions.

**Example:**

```
try:
    file = open('file.txt', 'r')
except FileNotFoundError:
    print("File not found.")
except OSError:
    print("OS error occurred.")
```

'FileNotFoundError' would never be caught if 'OSError' was caught before it.

**19. How can you handle deeply nested JSON structures in Python? Provide an example of reading, modifying, and writing such a JSON structure.**

**Answer:** This allows you to read, write, and modify data in deeply nested JSON structures by traversing the structure recursively.

**Example:**

```
import json

def modify_json(data):
    if isinstance(data, dict):
        for key, value in data.items():
            if key == 'target_key':
                data[key] = 'new_value'  # Modify the value
            modify_json(value)  # Recurse into nested
dictionaries
    elif isinstance(data, list):
        for item in data:
            modify_json(item)  # Recurse into lists

with open('deep_nested.json', 'r') as file:
    data = json.load(file)

modify_json(data)

with open('deep_nested_modified.json', 'w') as file:
    json.dump(data, file, indent=4)
```

**20. Can the finally clause modify the exception raised in the try block? Provide a code example that demonstrates this behavior.**

**Answer:** The exception raised in the 'try' block cannot be directly modified by the 'finally' clause, but it can be suppressed if it produces a new exception or, in the case of a function, returns a value.

```
def test():
    try:
        raise ValueError("Original exception")
    finally:
        return "Modified by finally"  # This will suppress the
original exception

print(test())  # Output: Modified by finally
```

**21. How would you efficiently search for a specific pattern in a large text file without loading the entire file into memory? What are the trade-offs of different approaches?**

**Answer: You can examine a large text file line by line or in chunks to look for a specific pattern:**

```
pattern = "search_term"

with open('large_file.txt', 'r') as file:
    for line in file:
        if pattern in line:
            print(line)
```

**Trade-offs:**

**Line-by-Line:** This method is memory-efficient, although it could take longer if the file contains a lot of lines.

**Chunk-Based:** While reading in larger chunks can make the process go faster, it also necessitates handling incomplete lines at chunk boundaries with caution.

Both methods work well with really large files because they don't load the complete file into memory.