8 File Handling

File handling is an essential aspect of programming, allowing applications to store, retrieve, and manipulate data persistently. Unlike working with in-memory data structures such as lists or dictionaries, files provide a way to store data permanently, ensuring it remains available even after the program stops running. In Python, file handling is a fundamental feature of the standard library, offering simple and powerful tools for reading, writing, and managing files efficiently.

Python provides built-in functions and modules that enable seamless interaction with files. Whether you need to read configuration settings, process large datasets, log system events, or store user-generated content, file handling plays a crucial role. Without file operations, many applications would be unable to save progress, transfer data between programs, or interact with external sources such as databases and APIs.

At its core, Python's file handling capabilities revolve around two primary types of files:

- **Text Files**: These files store data in a human-readable format, typically encoded in UTF-8 or ASCII. Examples include .txt, .csv, .json, and .xml files. Text files are widely used for storing logs, configuration files, structured data, and documents.
- **Binary Files**: These files store data in a format that is not directly human-readable, such as images, audio, video, and executable files. Common examples include .jpg, .png, .mp3, .exe, and .dat files. Working with binary files requires handling raw byte data instead of textual content.

Python provides a unified way to work with both text and binary files, allowing developers to open, read, write, and modify file contents with ease. The built-in open() function is at the heart of file operations, offering multiple modes for interacting with files based on the intended purpose.

Topics Covered in This Chapter: File Handling

- **Introduction to File Handling**: Understanding file handling, its importance, and the distinction between text and binary files.
- **Basic File Operations**: Using the open() function and understanding different file modes: read (r), write (w), append (a), exclusive creation (x), binary (b), text (t), and combined read-write modes (+).
- **Reading Files**: Reading file contents using read(), readline(), and readlines(), and iterating through files efficiently.
- **Writing Files**: Writing data with write() and writing multiple lines with writelines().
- **File Closing**: Importance of closing files using close() and best practices with the with statement for automatic file closure.
- **Handling Exceptions in File Operations**: Managing common errors such as FileNotFoundError, PermissionError, and IOError using try...except blocks.
- **File Path Handling**: Understanding absolute vs. relative paths, using os and pathlib for advanced path operations, and ensuring cross-platform compatibility.

Basic File Operations

File handling in Python revolves around the open() function, which allows reading from and writing to files efficiently. Whether you're working with text files or binary data, Python provides a range of modes to control how a file is accessed. Understanding these modes ensures that files are handled correctly, preventing accidental data loss or corruption.

Opening Files with the open() Function

To interact with a file, it must first be opened. Python provides the open() function, which takes two primary arguments:

1. The **file name** (including its path, if necessary).
2. The **mode** that specifies how the file should be accessed.

Basic Syntax

```
file = open("example.txt", "r")  # Open a file in read mode
```

This opens example.txt in **read mode** (r). If the file does not exist, an error occurs.

After performing file operations, it is essential to close the file using the close() method to free system resources.

```
file.close()
```

Alternatively, Python provides the **with statement**, which automatically handles file closure:

```
with open("example.txt", "r") as file:
   content = file.read()
```

With this approach, the file is automatically closed when the block finishes executing.

File Modes in Python

Python’s file handling system supports different modes, which determine how files are accessed. These modes can be used alone or combined to achieve specific functionality.

Read Mode (r)

Opens a file for reading. If the file does not exist, an error (FileNotFoundError) occurs.

```
with open("example.txt", "r") as file:
   content = file.read()
   print(content)
```

- Default mode if no mode is specified (open("example.txt") is equivalent to open("example.txt", "r")).
- The file must exist before opening.

Write Mode (w)

Opens a file for writing. If the file exists, its contents are **erased** before writing. If the file does not exist, it is created.

```
with open("example.txt", "w") as file:
    file.write("Hello, World!")
```

- Overwrites existing content.
- Creates a new file if it does not exist.

Exclusive Creation Mode (x)

Creates a new file but raises an error if the file already exists.

```
with open("newfile.txt", "x") as file:
    file.write("This file was just created!")
```

- Useful when you want to ensure a file does not get overwritten accidentally.
- If newfile.txt already exists, a FileExistsError is raised.

Append Mode (a)

Opens a file for writing but does not erase existing content. New data is added at the end of the file.

```
with open("example.txt", "a") as file:
    file.write("\nAppending this line.")
```

- If the file does not exist, it is created.
- Content is **appended**, not overwritten.

Binary Mode (b)

Used to read and write binary files (e.g., images, videos, executable files). Can be combined with other modes (rb, wb, ab).

```
with open("image.jpg", "rb") as file:
    binary_data = file.read()
```

- Works with non-text files such as images, PDFs, and audio files.

Writing binary files:

```
with open("copy.jpg", "wb") as file:
    file.write(binary_data)
```

Text Mode (t)

Text mode is the **default mode** in Python. It allows working with textual data. Can be combined with other modes (rt, wt, at).

```
with open("example.txt", "rt") as file:  # Equivalent to "r"
    content = file.read()
```

- Reads and writes text files, handling encoding and newline conversions automatically.

Binary mode (b) and text mode (t) can be combined with other modes (e.g., rb, wb, rt, wt), but they are **mutually exclusive**—you must choose either binary or text mode.

Read and Write Mode (+)

Enables both **reading and writing** to a file. Common variations include:

- r+: Read and write (file must exist, does not erase content).
- w+: Read and write (overwrites file if it exists, creates a new file if not).
- a+: Read and write (appends data if the file exists, creates a new file otherwise).

Example:

```
with open("example.txt", "r+") as file:
  content = file.read()
  file.write("\nNew line added.")
```

- Allows **reading and updating** the file at the same time.
- File pointer starts at the beginning unless a+ is used.

Summary of File Modes

Mode	Description
r	Read mode (file must exist)
w	Write mode (overwrites file, creates if missing)
x	Exclusive creation mode (fails if file exists)
a	Append mode (creates file if missing, does not overwrite)
b	Binary mode (used with r, w, or a)
t	Text mode (default, used with r, w, or a)
+	Read and write mode (combines with r, w, or a)

Choosing the Right File Mode

- **Use r** if you only need to read from a file without modifying it.
- **Use w** if you want to write new content, replacing any existing data.
- **Use a** if you need to add new content while preserving existing data.
- **Use x** to avoid accidental overwriting by ensuring the file does not already exist.
- **Use b** when working with binary files such as images, PDFs, or audio.

- **Use +** if you need to both read and write within the same file.

Conclusion

Understanding Python's file modes is crucial for effective file handling. The open() function provides flexibility in working with both text and binary files, while different modes allow for reading, writing, appending, and managing files securely. In the next section, we will explore how to read data from files using various methods, including reading entire files, reading line-by-line, and handling large files efficiently.

Reading Files

Once a file is opened in **read mode (r)**, Python provides several methods to read its contents efficiently. Depending on the size of the file and how the data needs to be processed, you can either read the entire file at once, read it line by line, or iterate through it using a loop.

Reading the Entire File with read()

The read() method reads the entire content of a file as a single string. This is useful when you need to process the whole file at once.

Example: Reading the Full Content of a File

```
with open("example.txt", "r") as file:

    content = file.read()

    print(content)  # Displays the entire file content
```

- If the file is **large**, reading it all at once may consume too much memory.
- The read() method returns an **empty string** ("") if the end of the file is reached.

Reading a Specific Number of Characters

The read(size) method allows reading only a specified number of characters.

```
with open("example.txt", "r") as file:

    partial_content = file.read(20)  # Reads only the first 20 
characters

    print(partial_content)
```

- This is useful for handling large files by reading them in chunks.
- Subsequent calls to read(size) continue from where the last read left off.

Reading Line by Line with readline() and readlines()

For structured text files, it is often more convenient to read the file **one line at a time** rather than loading everything at once.

Using readline() to Read One Line at a Time

The readline() method reads a **single line** from the file, including the newline (\n) character.

```
with open("example.txt", "r") as file:
    first_line = file.readline()
    second_line = file.readline()
    print("First Line:", first_line)
    print("Second Line:", second_line)
```

- Calling readline() repeatedly moves through the file line by line.
- When the end of the file is reached, readline() returns an **empty string** ("").

Using readlines() to Read All Lines into a List

The readlines() method reads all lines of a file and returns them as a **list of strings**, where each line is an element in the list.

```
with open("example.txt", "r") as file:
    lines = file.readlines()
    print(lines)
```

- Each line includes a newline character (\n).
- Useful when processing lines sequentially or storing them for later use.

If the file contains:

Apple

Banana

Cherry

The output will be:

['Apple\n', 'Banana\n', 'Cherry\n']

You can **strip newlines** using list comprehension:

```
cleaned_lines = [line.strip() for line in lines]
print(cleaned_lines)  # Output: ['Apple', 'Banana', 'Cherry']
```

Looping Through a File for Line-by-Line Processing

Instead of manually calling readline(), a more efficient way to **process each line in a file** is by iterating over the file object in a for loop.

Example: Reading a File Line by Line Using a Loop

```
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # Removes trailing newline characters
```

- The file object itself is an **iterator**, meaning it can be used directly in a loop.
- This approach is **memory efficient**, as it reads one line at a time instead of loading the entire file into memory.

- The strip() method ensures that trailing newline characters are removed.

Processing Lines with a Condition

```
with open("example.txt", "r") as file:
    for line in file:
        if "Error" in line:  # Process only lines containing "Error"
            print("Found an error log:", line.strip())
```

- Useful for **filtering specific lines** in log files or large datasets.

Choosing the Right Method for Reading Files

Method	Description	Best Use Case
read()	Reads the entire file as a single string	Small files that fit into memory
read(size)	Reads a specific number of characters	Reading files in chunks
readline()	Reads a single line at a time	When processing a file line by line
readlines()	Reads all lines into a list	When storing lines for later use
Looping through file	Iterates line by line efficiently	Processing large files without loading everything at once

Conclusion

Python provides multiple ways to read files depending on memory constraints and processing needs. The read() method retrieves the full content at once, while readline() reads one line at a time. The readlines() method stores all lines in a list, whereas iterating through the file object offers a memory-efficient way to process files line by line. In the next section, we will explore how to write data to files using different file modes and methods.

Writing Files

Writing data to files is a crucial aspect of file handling in Python. Whether you need to log system events, store processed data, or generate reports, Python provides efficient methods to write text or binary data to files. The write() method allows writing individual strings to a file, while writelines() enables writing multiple lines at once.

Writing to a File with write()

The write() method writes a string to a file. If the file does not exist, it is created. If the file exists, its content is **overwritten** when opened in write mode (w).

Example: Writing a Single Line to a File

```
with open("output.txt", "w") as file:

    file.write("Hello, World!\n")
```

- This code creates (or overwrites) output.txt and writes "Hello, World!\n" into it.
- The \n ensures that the next write operation starts on a new line.
- If write() is called multiple times, each call appends content sequentially but does **not** add newlines automatically.

Writing Multiple Strings Sequentially

```
with open("output.txt", "w") as file:

    file.write("First Line\n")

    file.write("Second Line\n")

    file.write("Third Line\n")
```

- Each call to write() adds text to the file but does **not** insert newlines unless explicitly included.

Writing Multiple Lines with writelines()

The writelines() method writes **a list of strings** to a file in one operation.

Example: Writing a List of Lines to a File

```
lines = ["Python is great!\n", "File handling is important.\n", 
"Writing multiple lines is easy.\n"]



with open("output.txt", "w") as file:

    file.writelines(lines)
```

- The list elements are written **exactly as they are**.
- Unlike write(), writelines() does **not** insert newline characters automatically. Ensure that each string in the list **ends with \n** to maintain proper line formatting.

Adding New Lines Manually

```
lines = ["Line 1", "Line 2", "Line 3"]

with open("output.txt", "w") as file:

    file.writelines(line + "\n" for line in lines)
```

- This ensures that each line appears correctly formatted in the file.

Writing Without Overwriting: Using Append Mode (a)

If a file is opened in append mode (a), data is added **at the end** of the file without overwriting existing content.

Example: Appending Data to an Existing File

```
with open("output.txt", "a") as file:

    file.write("This is an additional line.\n")
```

- The new content is **added at the end** of the file.
- If the file does not exist, it is created.

Appending Multiple Lines

```
new_lines = ["Appending line 1\n", "Appending line 2\n"]

with open("output.txt", "a") as file:

    file.writelines(new_lines)
```

- The existing content remains intact, and the new lines are added at the bottom.

Choosing the Right Method for Writing

Method	Description	When to Use
write(string)	Writes a single string to the file	Writing a small amount of data
writelines(list)	Writes a list of strings	Writing multiple lines efficiently
w mode	Overwrites the file if it exists	Creating new content from scratch
a mode	Appends content to the end of the file	Updating log files or appending new data

Conclusion

Python provides flexible methods for writing to files, whether it’s writing single lines using write() or multiple lines using writelines(). The **write mode (w)** overwrites the file, while **append mode (a)** preserves existing content. Understanding these techniques allows efficient data storage, logging, and report generation. In the next section, we will explore file handling best practices, including error handling and closing files properly.

File Closing

Properly closing files after reading or writing is a crucial part of file handling in Python. When a file is open, the operating system allocates resources to manage it. If files are not closed correctly, it can lead to resource leaks, unexpected behavior, and issues in multi-user or multi-threaded environments. Python provides both **manual file closing** using close() and an **automatic approach** using the with statement.

The Importance of Closing Files with close()

The close() method is used to **release system resources** associated with an open file. If a file remains open for an extended period, it may lock access to other programs or cause memory issues.

Example: Manually Closing a File

```
file = open("example.txt", "w")

file.write("This is a test file.")

file.close()  # Closing the file manually
```

- The file remains open until close() is explicitly called.
- If the program crashes or an exception occurs before close(), the file may remain open indefinitely.

Risks of Not Closing Files

If a file is not properly closed:

- It may cause excessive memory consumption in large applications.
- Other programs may not be able to access or modify the file.
- Data might not be written completely, leading to corruption or incomplete files.

Managing Resources Effectively with with Statement

To avoid the risk of forgetting to close a file, Python provides the **with statement**, which **automatically handles file closure**, even if an error occurs.

Example: Using with for Automatic File Closure

```
with open("example.txt", "w") as file:

    file.write("This file is managed using 'with'.")
```

- The file is **automatically closed** once the block inside with is executed.
- Even if an exception occurs inside the block, Python ensures that the file is closed properly.

Why Use with Instead of close()?

- Eliminates the need to manually call close().
- Prevents resource leaks, ensuring files do not remain open longer than necessary.
- Makes code cleaner and easier to read.

Best Practices for File Closing

Method	**Description**	**When to Use**
close()	Manually closes a file after use	When managing files explicitly (less recommended)

Method	Description	When to Use
with open()	Automatically closes the file after exiting the block	Always preferred for safe and efficient file handling

Conclusion

Closing files is essential to prevent resource leaks and ensure smooth file operations. While close() can be used to manually close a file, the **with statement is the recommended approach** as it guarantees automatic file closure. This improves code reliability and prevents common file handling issues. In the next section, we will explore file handling exceptions and how to manage errors effectively.

Handling Exceptions in File Operations

File operations in Python can sometimes lead to errors due to issues like missing files, lack of permissions, or corrupted data. If these errors are not handled properly, they can cause the program to crash. Python provides **exception handling mechanisms** to gracefully manage such situations, ensuring that the program does not terminate unexpectedly.

This section covers **common file handling errors** and how to use **try...except blocks** to handle them effectively.

Common File Handling Errors

Several exceptions may occur when working with files. The most common ones include:

1. FileNotFoundError

Occurs when trying to open a file that does not exist.

Example:

```
try:

    file = open("nonexistent.txt", "r")  # File does not exist

    content = file.read()

    file.close()

except FileNotFoundError:

    print("Error: The file was not found.")
```

Output:

```
Error: The file was not found.
```

2. PermissionError

Occurs when attempting to access a file without the necessary permissions (e.g., trying to write to a read-only file).

Example:

```
try:
    file = open("/protected_folder/restricted.txt", "w")  # Restricted access
    file.write("Trying to write to a protected file.")
    file.close()
except PermissionError:
    print("Error: You do not have permission to write to this file.")
```

3. IOError

A general error that occurs when an **input/output operation fails**. It may happen due to hardware issues, network errors (if dealing with remote files), or disk corruption.

Example:

```
try:
    with open("example.txt", "r") as file:
        data = file.read()
except IOError:
    print("Error: An issue occurred while reading the file.")
```

4. IsADirectoryError

Occurs when trying to open a directory as if it were a file.

Example:

```
try:
    file = open("some_directory", "r")  # Trying to open a directory as a file
except IsADirectoryError:
    print("Error: Expected a file but found a directory.")
```

5. EOFError (End of File Error)

Raised when trying to read beyond the end of a file.

Example:

```
try:
    file = open("empty.txt", "r")  # An empty file
    print(file.read())
except EOFError:
    print("Error: Reached end of file unexpectedly.")
```

Using try...except for Error Handling

To prevent the program from crashing when a file operation fails, **try...except blocks** should be used.

Example: Handling Multiple File Errors

```
try:
    with open("sample.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
    print("Error: The file does not exist.")
except PermissionError:
    print("Error: You do not have permission to access this file.")
except IOError:
```

print("Error: A general input/output error occurred.")

- If the file is missing, FileNotFoundError will be caught.
- If access is restricted, PermissionError will be caught.
- Any other I/O-related error will trigger the IOError exception.

Conclusion

Handling file exceptions is essential for writing robust Python programs. Errors such as FileNotFoundError, PermissionError, and IOError are common, but **using try...except blocks ensures that the program continues running smoothly** even when issues arise. The finally block can be used to guarantee file closure, preventing resource leaks. **By following best practices, developers can build reliable file-handling mechanisms that gracefully manage errors and system resources.**

File Path Handling

When working with files in Python, correctly handling file paths is essential to ensure that your code works consistently across different operating systems and environments. **File paths specify the location of a file or directory on a system**, and handling them properly helps avoid errors related to missing files, incorrect directories, or operating system inconsistencies.

This section covers the difference between **absolute and relative paths**, how to use **the os and pathlib modules** for advanced file path operations, and techniques for ensuring **cross-platform compatibility**.

Absolute vs. Relative Paths

Absolute Paths

An **absolute path** specifies the full location of a file or directory from the root of the file system. It always points to the same location, no matter where the script is executed.

Example of an absolute path:

- **Windows:** "C:\Users\Alice\Documents\file.txt"
- **Mac/Linux:** "/home/alice/Documents/file.txt"

Example in Python:

```
file_path = "/home/user/documents/data.txt"  # Absolute path (Linux/Mac)

file_path = "C:\\Users\\User\\Documents\\data.txt"  # Absolute path (Windows)
```

Relative Paths

A **relative path** specifies the location of a file **relative to the current working directory (CWD)**. This makes the code more flexible and portable.

Example of a relative path:

- "data/file.txt" → Refers to file.txt inside the data folder in the current directory.

Example in Python:

```
file_path = "data/report.txt"  # Relative path to a file inside 'data' folder
```

Getting the Current Working Directory

To determine the current directory from which the script is being run, use:

```
import os


print("Current Working Directory:", os.getcwd())  # Returns the full absolute path
```

If your script is located in /home/user/project/ and the file data.txt is inside project, using "data.txt" as a relative path would refer to /home/user/project/data.txt.

Using os and pathlib for File Paths

Python provides two modules for working with file paths:

- **os.path**: Provides functions for working with file and directory paths.
- **pathlib**: Introduced in Python 3.4, it offers an object-oriented way to handle file paths.

Joining Paths Dynamically (os.path.join)

Instead of manually concatenating paths, os.path.join() ensures proper formatting.

```
import os

```

```
folder = "documents"

filename = "report.txt"

file_path = os.path.join(folder, filename)


print(file_path)  # Output: documents/report.txt (Linux/Mac) OR
documents\report.txt (Windows)
```

Checking If a File or Directory Exists (os.path.exists)

Before accessing a file, check if it exists to avoid errors.

```
file_path = "data.txt"


if os.path.exists(file_path):

    print("File exists")

else:

    print("File not found")
```

Extracting Parts of a Path

```
file_path = "/home/user/documents/report.txt"


print("Directory:", os.path.dirname(file_path))  #
/home/user/documents

print("File name:", os.path.basename(file_path))  # report.txt

print("File extension:", os.path.splitext(file_path)[1])  # .txt
```

Creating Paths with pathlib.Path

```
from pathlib import Path


file_path = Path("documents") / "report.txt"

print(file_path)  # Output: documents/report.txt
```

Getting the Absolute Path

```
print(file_path.resolve())  # Returns the absolute path
```

Checking File Existence

```
if file_path.exists():

    print("File exists")

else:
    print("File does not exist")
```

Extracting File Components

```
print("Directory:", file_path.parent)  # documents

print("File name:", file_path.name)  # report.txt

print("File extension:", file_path.suffix)  # .txt
```

Creating Directories

```
Path("new_folder").mkdir(exist_ok=True)

print("Directory created!")

The mkdir() method ensures that a directory is created only if it
does not already exist.
```

Handling Cross-Platform File Paths

Different operating systems use different path formats:

- **Windows** uses backslashes (C:\Users\User\file.txt).
- **Linux/Mac** uses forward slashes (/home/user/file.txt).

Python's os and pathlib modules handle these differences automatically.

Using os.path for Cross-Platform Paths

```
import os


file_path = os.path.join("folder", "subfolder", "file.txt")

print(file_path)  # Windows: folder\subfolder\file.txt | Mac/Linux:
folder/subfolder/file.txt
```

Using pathlib.Path for Cross-Platform Paths

```
from pathlib import Path


file_path = Path("folder") / "subfolder" / "file.txt"

print(file_path)  # Works correctly on all OS
```

Using os.name to Detect OS

```
import os


if os.name == "nt":  # Windows

    print("Running on Windows")
```

```
elif os.name == "posix":  # Linux/Mac
    print("Running on Linux/Mac")
```

Using sys.platform for More Detailed OS Detection

```
import sys


if sys.platform.startswith("win"):
    print("Windows detected")
elif sys.platform.startswith("linux"):
    print("Linux detected")
elif sys.platform.startswith("darwin"):
    print("Mac detected")
```

Conclusion

Handling file paths correctly is crucial for **writing portable and error-free code**. Python provides both the os module (for traditional path handling) and the pathlib module (for an object-oriented approach). Understanding the difference between **absolute and relative paths**, using **cross-platform path handling**, and utilizing **built-in tools for checking file existence and manipulating paths** will help in managing files effectively across different operating systems.

Summary

File handling is a crucial aspect of programming that enables applications to store, retrieve, and manipulate data persistently. Unlike in-memory data structures, files allow data to remain accessible even after a program has stopped running. Python provides built-in support for working with files, making it easy to read, write, and manage both text and binary data efficiently. This chapter explores the core principles of file handling, from basic file operations to handling errors and working with file paths.

At the heart of file handling is the open() function, which allows files to be accessed in different modes. The read mode (r) is used to retrieve file contents, the write mode (w) creates or overwrites files, the append mode (a) adds data to an existing file without erasing its content, and the exclusive creation mode (x) ensures a file is created only if it does not already exist. Additionally, Python supports binary mode (b) for working with non-text files such as images and videos, and text mode (t), which is the default. Combining these modes with + enables simultaneous reading and writing.

Reading files in Python can be done in multiple ways depending on the size of the file and how the data needs to be processed. The read() method retrieves the entire file as a string, while readline() reads a single line at a time, making it useful for sequential processing. The readlines() method returns all lines as a list, and iterating over a file object using a loop is often the most memory-efficient way to process large files. Writing to files follows a similar approach—write() allows adding text to a file, whereas writelines() writes multiple lines efficiently.

Properly closing files is essential to prevent resource leaks and ensure data integrity. The close() method releases system resources, but the preferred approach is to use the with statement, which automatically closes the file when it is no longer needed. This ensures that files are properly handled, even if an error occurs.

Errors are common in file operations, and handling them effectively prevents crashes and unexpected behavior. Some frequent file-related errors include FileNotFoundError, which occurs when trying to read a non-existent file, PermissionError, which arises when attempting to access restricted files, and IOError, which covers general input/output issues. Using try...except blocks helps catch these exceptions and ensures that the program continues running smoothly.

Managing file paths correctly is another important aspect of file handling, especially when working across different operating systems. Python differentiates between absolute paths, which specify the complete location of a file, and relative paths, which locate files in relation to the current working directory. The os and pathlib modules provide tools for handling file paths dynamically, ensuring compatibility across Windows, macOS, and Linux. Functions like os.path.join() help construct paths in a platform-independent way, while pathlib offers an object-oriented approach for path manipulation.

By the end of this chapter, you will have a strong understanding of how to read, write, and manage files effectively in Python. You will also be equipped to handle errors gracefully, use best practices for closing files, and work with file paths in a cross-platform manner. Mastering these concepts will allow you to build applications that can store and process data efficiently, making file handling a fundamental skill for any Python programmer.

Exercises

1. **Reading a File** – Write a Python program that reads the contents of a text file called sample.txt and prints it to the console. Ensure that the program handles the case where the file does not exist gracefully.
2. **Writing to a File** – Create a program that writes a list of strings to a file named output.txt. Each string should be written on a new line. Then, reopen the file and print its contents to verify the writing process.
3. **Appending Data to a File** – Modify the previous program to append new lines to the file instead of overwriting its contents. Ensure that existing data is preserved.
4. **Reading a File Line by Line** – Implement a script that reads a file called log.txt line by line and prints each line to the console. Use a loop to process large files efficiently.
5. **Counting Words in a File** – Write a function that takes a filename as input and returns the number of words in the file. Test it by providing a text file with sample content.
6. **Checking File Existence** – Create a program that checks if a file exists before attempting to read it. If the file does not exist, display an appropriate message instead of raising an error.
7. **Copying a File** – Write a script that copies the contents of a file called source.txt to a new file called destination.txt. Ensure that it works for both text and binary files.
8. **Handling File Paths with pathlib** – Write a program that takes a relative file path as input, converts it to an absolute path using the pathlib module, and prints it to the console.

9. **Finding and Replacing Text in a File** – Write a Python script that searches for a specific word in a text file and replaces all occurrences with another word. The modified content should be saved to a new file, ensuring that the original file remains unchanged.
10. **Handling Exceptions in File Operations** – Implement a robust file handling function that attempts to read a file, handles errors such as FileNotFoundError and PermissionError, and prints appropriate error messages.