Python File Handling

Premanand S

Assistant Professor
School of Electronics Engineering
Vellore Institute of Technology
Chennai campus

premanand.s@vit.ac.in

November 12, 2024

What is File Handling?

- **Definition**: File handling is the process of performing operations such as reading from, writing to, and managing files within a program.
- **Purpose**: Enables data storage, retrieval, and manipulation for persistent data access.
- Commonly used in applications for:
  - Data storage
  - Configuration files
  - Logging and reporting

Why is File Handling Important?

- **Data Persistence**: Files enable data to be saved and accessed later.
- **Data Exchange**: Allows different applications to share information.
- **Efficient Manipulation**: Read, write, update, and delete data directly.

Common Use Cases for File Operations

- **Data Storage**: Save data like user preferences and app settings.
- **Configuration Files**: Store settings, paths, API keys, and more.
- **Logging**: Record events, errors, and debug information.
- **Data Serialization**: Store data in JSON, XML, CSV for analysis.
- **Backup and Recovery**: Save files as backups for system recovery.
- **Web Development**: Handle file uploads or session data.
- **Database Simulation**: Use files as a lightweight alternative to databases.

File Operations in Python

Python provides several operations to work with files:

- **Create**: Make a new file if it doesn't already exist.
- **Open**: Open a file to perform read or write operations.
- **Read**: Extract data from a file.
- **Write**: Add or modify data in a file.
- **Append**: Add data to the end of an existing file.
- **Close**: Free up system resources by closing the file.

File Modes in Python

Different modes allow for specific file operations:

- **'r'** - Read Mode: Opens the file for reading. Raises an error if the file doesn't exist.
- **'w'** - Write Mode: Opens the file for writing, creating a new file if it doesn't exist, and truncating the file if it exists.
- **'a'** - Append Mode: Opens the file for appending new data to the end without truncating it. Creates a new file if it doesn't exist.
- **'r+'** - Read and Write Mode: Opens the file for both reading and writing. Raises an error if the file doesn't exist.
- **'w+'** - Write and Read Mode: Opens the file for reading and writing, creating a new file if it doesn't exist, and truncating it if it does.
- **'a+'** - Append and Read Mode: Opens the file for reading and appending. Creates a new file if it doesn't exist.

Choosing the Right File Mode

- **'r'**: Use when you only need to read the contents of an existing file.
- **'w'**: Use when you need to create a new file or overwrite an existing one.
- **'a'**: Use when you want to add new data to the end of an existing file.
- **'r+'**: Use when you need to both read from and write to an existing file.
- **'w+'**: Use when you need to create a new file or overwrite an existing one, but also need to read from it.
- **'a+'**: Use when you want to read from an existing file and append new data to the end.

Summary of File Modes

Mode	Operation	File Exists	File Doesn't Exist
r	Read only	Opens file	Raises Error
w	Write only	Truncates file	Creates new file
a	Append only	Opens file	Creates new file
r+	Read/Write	Opens file	Raises Error
w+	Write/Read	Truncates file	Creates new file
a+	Append/Read	Opens file	Creates new file

Creating a File

- When a file is opened in `'w'` (write) or `'a'` (append) mode, Python checks if the file exists.
- If the file does not exist, Python automatically creates a new file.
- Example:

Example (Python Code)

```
file = open("newfile.txt", "w")
file.write("This is a new file.")
file.close()

#This creates "newfile.txt" if it doesn't already exist.
```

Opening a File: Using the `open()` Function

- Syntax: `open(filename, mode)`
- **Parameters:**
  - `filename`: The name of the file to open.
  - `mode`: The mode in which the file is opened (e.g., `'r'`, `'w'`, `'a'`, `'r+'`).
- **Returns**: A file object.

Example (Python Code)

Example: Opening Files in Different Modes

Read Mode (''r'')

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

Opens the file for reading. Raises an error if the file does not exist.

Write Mode (''w'')

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

Opens the file for writing, truncating the file if it exists, or creating it if it does not exist.

Append Mode (''a'')

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

Opens the file for appending data. Creates the file if it does not exist.

Example: Opening Files in Read and Write Mode

Read and Write Mode (''r+'')

```
file = open("example.txt", "r+")
```

Opens the file for both reading and writing. Raises an error if the file does not exist.

Write and Read Mode (''w+'')

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

Opens the file for reading and writing. Creates a new file if it does not exist, or truncates it if it does.

Append and Read Mode (''a+'')

```
file = open("example.txt", "a+")
```

Opens the file for reading and appending. Creates a new file if it does not exist.

Closing the File

- It is good practice to close a file after operations to free up system resources.
- Use `file.close()` to close the file.
- Example:

Example (Python Code)

```
file = open("example.txt", "r")
...
file.close()
```

Reading Files in Python

Python offers several methods to read data from files:

- `.read()`: Reads the entire file content as a single string.
- `.readline()`: Reads a single line from the file.
- `.readlines()`: Reads all lines in the file and returns a list of strings.

Using `.read()` Method

- `.read()` reads the entire file as a single string.
- Useful when the file size is manageable in memory.
- Example:

Code

```
file = open("example.txt", "r")
content = file.read()
print(content)
file.close()
```

Using `.readline()` Method

- `.readline()` reads one line at a time.
- Useful for reading files line-by-line when file size is large.
- Example:

Code

```
file = open("example.txt", "r")
line = file.readline()
while line:
  print(line.strip())
  line = file.readline()
file.close()
```

Using `.readlines()` Method

- `.readlines()` reads all lines at once and returns them as a list of strings.
- Each element in the list represents one line in the file.
- Example:

Code

```
file = open("example.txt", "r")
lines = file.readlines()
for line in lines:
  print(line.strip())
file.close()
```

Iterating Through Files with a Loop

- Files can be read line-by-line using a `for` loop, which is memory-efficient.
- This is especially useful for large files.
- Example:

Code

```
file = open("example.txt", "r")
for line in file:
  print(line.strip())
file.close()
```

Handling Large Files Efficiently

- Reading the entire file at once can be memory-intensive for large files.
- Use `.readline()` or `for line in file` to process one line at a time.
- Example: Using `with` statement to ensure proper file closure.

Code

```
with open("largefile.txt", "r") as file:
  for line in file:
    process(line)
```

The `with` statement automatically closes the file after processing.

Writing to Files in Python

Python provides multiple methods for writing to files:

- `.write()`: Writes a single string to the file.
- `.writelines()`: Writes a list of strings to the file.

Each method can be used with different file modes for specific purposes.

Using `.write()` Method

- `.write()` is used to write a single string to a file.
- The method returns the number of characters written.
- Example:

Code

```
file = open("example.txt", "w")
file.write("Hello, World!")
file.close()
```

This code creates or overwrites `example.txt` with "Hello, World!".

Using `.writelines()` Method

- `.writelines()` writes multiple lines to a file from a list of strings.
- Each string in the list is written without automatically adding line breaks.
- Example:

Code

```
file = open("example.txt", "w")
lines = ["Hello, World!
n", "Welcome to File Handling.
n"]
file.writelines(lines)
file.close()
```

This code writes each string from `lines` to `example.txt`.

Overwriting vs. Appending

- Opening a file in `'w'` mode (write mode):
  - Overwrites the existing content.
  - Creates a new file if it does not exist.
- Opening a file in `'a'` mode (append mode):
  - Adds new data to the end of the file.
  - Preserves existing content.

Example: Overwriting vs. Appending

Overwriting with 'w' Mode

```
file = open("example.txt", "w")
file.write("This will overwrite the file.")
file.close()
```

Appending with 'a' Mode

```
file = open("example.txt", "a")
file.write("This text will be added to the file.")
file.close()
```

Using 'w' mode clears the file, while 'a' mode adds to the end.

Appending to Files in Python

- The append mode (`'a'`) allows data to be added at the end of the file.
- When a file is opened in `'a'` mode:
  - Existing content is not erased.
  - New data is added after the current content.
- If the file does not exist, `'a'` mode creates a new file.

Example: Using Append Mode

- Here's how to add new content without deleting existing content:

Code

```
file = open("example.txt", "a")
file.write("This is an appended line.
n")
file.close()
```

- This code opens `example.txt` in `'a'` mode.
- The line "This is an appended line." will be added at the end.

Append Mode: Key Points

- Ideal for adding logs, additional entries, or new data over time.
- 'a' mode preserves all previous data in the file.
- A new line can be appended each time without modifying existing content.

Closing Files in Python

- After performing file operations, it's crucial to close the file to:
  - Free up system resources.
  - Ensure that all changes are saved and the file is properly closed.
  - Avoid potential file corruption or data loss.

Using `close()` Method

- `file.close()` is used to close an open file after operations.
- Always close the file after you are done with it to prevent resource leakage.

Code

```
file = open("example.txt", "w")
file.write("This is a test file.")
file.close()
```

This code writes to `example.txt` and then closes the file.

The `with` Statement

- The `with` statement simplifies file handling by automatically closing the file after operations are completed.
- No need to explicitly call `close()`.
- Ensures that the file is properly closed even if an exception occurs.

Using the `with` Statement for File Handling

Code Example

```
with open("example.txt", "w") as file:
 file.write("This will automatically close the file.")
 File is automatically closed when the block ends
```

- The file is opened and closed automatically without needing `file.close()`.
- This is a recommended practice as it ensures the file is closed even if an error occurs within the block.

File Cursor Management in Python

- The file cursor (or file pointer) is the position within the file where the next read or write operation will occur.
- When reading a file, the cursor moves forward, and when writing, the cursor moves forward after each write operation.
- Understanding cursor management is crucial for random access file operations.

Using the `tell()` Method

- `tell()` returns the current position of the file cursor.
- It helps to determine where in the file you are while performing read or write operations.

Code Example

```
file = open("example.txt", "r")
file.read(10)
print(file.tell())
file.close()
```

- In this example, after reading 10 characters, `tell()` will return the position of the cursor, which will be 10.

Using the `seek()` Method

- `seek()` is used to move the file cursor to a specific position in the file.
- Syntax: `file.seek(offset, whence)`
  - `offset`: The number of bytes to move the cursor.
  - `whence`: (optional) The reference point from where the offset is applied (default is `0` for the beginning).

Example: Using `seek()` Method

Code Example

```
file = open("example.txt", "r")
file.seek(5)
print(file.read(10))
file.close()
```

- This code moves the cursor 5 bytes from the beginning and then reads the next 10 characters.
- The result will be the characters starting from the 6th byte in the file.

Use Cases of `seek()` Method

- `seek()` is commonly used for:
  - Random access file reading and writing.
  - Rewinding a file (move cursor to the beginning using `seek(0)`).
  - Navigating through large files without loading the entire content into memory.

Error Handling in File Operations

- When working with files, various errors can occur, such as missing files, permission issues, or file access errors.
- Proper error handling is essential to avoid program crashes and provide meaningful error messages.
- Common errors in file handling include:
  - `FileNotFoundError`
  - `IOError`
  - `PermissionError`

Using try-except for Error Handling

- The try-except block allows you to handle exceptions gracefully and prevent crashes.
- If an error occurs in the try block, the code in the except block will be executed.

Code Example

```
try:
    file = open("non_existent_file.txt", "r")
except FileNotFoundError:
    print("File not found!  Please check the file path.")
except IOError:
    print("An IOError occurred.  Please check the file
permissions.")
finally:
    print("Execution complete.")
```

- In this example, if the file is not found, a message will be printed, and the program will continue without crashing.

Tips for Error-Free File Handling

- Always check if the file exists before attempting to read:
  - Use the `os.path.exists()` or `os.path.isfile()` method to check if the file exists.
- Use the `with` statement to handle files:
  - It automatically handles closing the file, reducing the risk of file access errors.
- Be cautious about file permissions, especially when writing files or accessing system files.
- Catch multiple exceptions in a single `except` block to handle various error types.

Example: Handling Multiple Errors

Code Example

```
import os
try:
   if not os.path.exists("file.txt"):
     raise FileNotFoundError("File does not exist.")
   file = open("file.txt", "r")
except FileNotFoundError as e:
   print(e)
except IOError as e:
   print("IOError:  ", e)
finally:
   print("Finished handling the file.")
```

- This example checks if the file exists first and raises a custom error if not, handling both `FileNotFoundError` and `IOError`.