Python File Handling: Complete Guide with Methods and Context Managers

What is File Handling in Python?

File handling refers to the process of working with files: opening them, reading their content, writing to them, and managing file operations. It allows Python programs to interact with data stored on disk, such as text files, logs, configuration files, or any external data sources.

Uses of File Handling

- Reading input data from files.
- Saving output or program results to files.
- Modifying or updating existing files.
- Managing logs or user data.
- Sharing data between programs or platforms.

Common File Handling Methods in Python

1. `open(filename, mode)`

Open a file and return a file object. Modes include:

Mode	Description
`"r"`	Read (default; file must exist)
`"w"`	Write (creates or overwrites)
`"a"`	Append (creates or writes at end)
`"b"`	Binary mode (used with other modes)
`"r+"`	Read and write (file must exist)
`"w+"`	Write and read (creates or overwrites)

`"a+"`	Append and read (creates if missing)

Example:

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

2. `read()`

Reads the entire content of the file as a string.

```
content = file.read()
print(content)
```

3. `readline()`

Reads a single line (including the newline character).

```
line = file.readline()
print(line)
```

4. `readlines()`

Reads all lines and returns them as a list of strings.

```
lines = file.readlines()
print(lines)
```

5. `write(string)`

Writes a string to the file. Overwrites existing content if in `"w"` mode.

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

6. `writelines(list_of_strings)`

Writes a list of strings to the file. **Does not** add newline characters automatically.

```
file.writelines(["First line\n", "Second line\n"])
```

7. `close()`

Closes the file and releases system resources.

```
file.close()
```

8. `seek(position)`

Moves the cursor to a specified byte position.

```
file.seek(0)  # Go to start of the file
```

9. `tell()`

Returns the current byte position of the cursor.

```
position = file.tell()
print(position)
```

10. `truncate(size)`

Resizes the file to a specified size (in bytes). Content beyond is discarded.

```
file.truncate(5)  # Keeps only first 5 bytes
```

11. `flush()`

Forces the buffer to write data to disk immediately.

```
file.flush()
```

Context Manager: Safely Handling Files with `with open(...) as`

Manually opening and closing files can lead to issues if your program crashes or raises exceptions before closing the file, resulting in resource leaks. Python's **context manager** (`with` statement) handles this safety concern by automatically opening and closing the file.

Why Use a Context Manager?

- Ensures files are always properly closed.
- Cleaner, more readable code.
- Handles exceptions gracefully without leaking resources.

Example of Using Context Manager

```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
# File is automatically closed here, even if an error occurs in the block.
```

This approach calls the file's internal `__enter__()` method when entering the block and `__exit__()` method when exiting, taking care of setup and cleanup automatically.

Custom Context Manager Example

You can create your own context managers for custom resource management using the `contextlib` module or by defining `__enter__` and `__exit__` methods.

```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
    def __enter__(self):
```

```
        self.file = open(self.filename, self.mode)
        return self.file
    def __exit__(self, exc_type, exc_value, traceback):
        self.file.close()

with FileManager("test.txt", "w") as f:
    f.write("Hello")
```

Quick Reference: Example Using Multiple Methods

```
with open("example.txt", "w") as file:
    file.write("Line 1\n")
    file.writelines(["Line 2\n", "Line 3\n"])
    file.flush()

with open("example.txt", "r") as file:
    print("--- Full Content ---")
    print(file.read())

with open("example.txt", "r") as file:
    file.seek(0)
    print("--- First Line ---")
    print(file.readline(), end='')

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

Summary

Method	Description
`open()`	Open a file with a specified mode
`read()`	Read entire file content
`readline()`	Read one line at a time
`readlines()`	Read all lines into a list
`write()`	Write a string to the file
`writelines()`	Write a list of strings
`close()`	Close the file
`seek()`	Move file cursor
`tell()`	Get current cursor position
`truncate()`	Resize file
`flush()`	Flush buffer to disk
`with open... as`	Context manager for automatic open/close

Feel free to ask if you want code examples for any specific method or concept!