Python File Handling

A comprehensive guide for beginners and transitioning programmers

1. Introduction to File Handling

Why File Handling is Essential

File handling is one of the most fundamental aspects of programming. It allows your programs to interact with the permanent storage on a computer system, making it possible to:

- Store data permanently between program executions
- Process large datasets that won't fit in memory
- Import and export data to/from other applications
- Configure application behavior through settings files
- Log information for debugging and auditing
- Share information between different programs or systems

For programmers coming from other languages (like C/C++, Java, etc.), Python offers a much more streamlined approach to file handling with powerful, high-level abstractions.

Types of Files in Python

In Python, we primarily work with two types of files:

1. Text Files

- **Content**: Human-readable text (characters, strings)
- **Access mode**: Text mode ('t')
- **Line endings**: Automatically translated to platform-specific conventions
- **Common formats**: .txt, .csv, .html, .py, .json, .xml
- **Encoding considerations**: Need to specify encoding (UTF-8, ASCII, etc.)

2. Binary Files

- **Content**: Raw bytes (not human-readable)
- **Access mode**: Binary mode ('b')
- **Line endings**: No translation (preserved as-is)
- **Common formats**: .jpg, .png, .pdf, .mp3, .xlsx, .zip

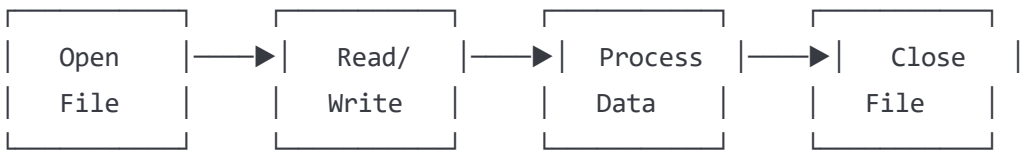
- **Encoding considerations**: No text encoding (raw bytes)

Understanding the difference is crucial because:

- Text files require encoding/decoding between bytes and text
- Binary files are processed as raw bytes without interpretation
- Operations on binary files preserve exact bit patterns

Basic File Operations Flowchart

The general pattern for file operations follows this sequence:



In Python, this is typically condensed using context managers, which we'll explore later:

```python
with open('file.txt', 'r') as file:  # Open
    data = file.read()               # Read
    # Process data                   # Process
                                     # Close (automatic)
```

2. Opening and Closing Files

The `open()` Function in Detail

The built-in `open()` function is Python's gateway to file operations:

```python
file_object = open(file, mode='r', buffering=-1, encoding=None,
                   errors=None, newline=None, closefd=True, opener=None)
```

Essential Parameters:

1. `file` (required): Path to the file (string or path-like object)

```python
# Relative path
file = open('data.txt')

# Absolute path
file = open('C:/Users/username/Documents/data.txt')  # Windows
file = open('/home/username/documents/data.txt')     # Unix/Linux/Mac

# Using pathlib (modern approach)
from pathlib import Path
file = open(Path.home() / 'documents' / 'data.txt')
```

2. `mode` (optional): Specifies file access mode (default is 'r')

File Modes Explained

Mode	Description	Creates New File?	Truncates Existing File?	Position
'r'	Read (default)	No	No	Beginning
'w'	Write	Yes	Yes	Beginning
'a'	Append	Yes	No	End
'x'	Exclusive creation	Yes (error if exists)	N/A	Beginning
'r+'	Read and write	No	No	Beginning
'w+'	Read and write	Yes	Yes	Beginning
'a+'	Read and append	Yes	No	End for writing, beginning for reading

Mode Modifiers:

- **'t'**: Text mode (default)
- **'b'**: Binary mode

Mode Combinations:

python

```python
# Common combinations
file = open('data.txt', 'r')     # Read text (most common)
file = open('data.txt', 'w')     # Write text, truncate if exists
file = open('data.bin', 'rb')    # Read binary
file = open('data.bin', 'wb')    # Write binary
file = open('log.txt', 'a')      # Append text
file = open('data.txt', 'r+')    # Read and write text
```

Comparing with C-style File Opening

For those familiar with C programming, here's a comparison:

C Function	Python Equivalent	Description
`fopen("file.txt", "r")`	`open("file.txt", "r")`	Open for reading
`fopen("file.txt", "w")`	`open("file.txt", "w")`	Open for writing
`fopen("file.txt", "a")`	`open("file.txt", "a")`	Open for appending
`fopen("file.txt", "r+")`	`open("file.txt", "r+")`	Open for reading and writing
`fopen("file.bin", "rb")`	`open("file.bin", "rb")`	Open binary file for reading

Additional Open Parameters

3. `encoding`: Specifies the text encoding (for text files)

   python

   ```python
   # Common encodings
   file = open('data.txt', encoding='utf-8')     # Unicode (recommended)
   file = open('legacy.txt', encoding='ascii')   # ASCII
   file = open('windows.txt', encoding='cp1252') # Windows-specific
   ```

4. `errors`: How encoding/decoding errors are handled

   python

   ```python
   # Error handling strategies
   file = open('data.txt', encoding='utf-8', errors='strict')    # Default: raise exception
   file = open('data.txt', encoding='utf-8', errors='ignore')    # Skip problematic characters
   file = open('data.txt', encoding='utf-8', errors='replace')   # Replace with replacement ch
   ```

5. `newline`: Controls how universal newlines work

```python
# Newline handling
file = open('data.txt', newline=None)    # Default: translate to platform-specific
file = open('data.txt', newline='')      # No translation
file = open('data.txt', newline='\n')    # Specify exact newline character
```

The `with` Statement (Context Manager)

What is a Context Manager?

A context manager in Python is a special construct that handles the setup and teardown of resources automatically. For programmers coming from C, this is similar to how you might use functions to ensure proper resource initialization and cleanup, but much more elegant.

In C, you might handle a file like this:

```c
FILE *file = fopen("filename.txt", "r");
if (file == NULL) {
    // Handle error
    return -1;
}

// Process the file
char buffer[100];
fgets(buffer, 100, file);
// More operations...

// Don't forget to close the file!
fclose(file);
return 0;
```

Notice how you must manually:

1. Open the file
2. Check for errors
3. Process the file
4. Explicitly close the file

Problems with this approach:

- If an error occurs between opening and closing, you might forget to close the file
- If the function has multiple return points, you need to close the file at each one
- It's easy to forget the closing step

Python's Solution: The `with` Statement

The recommended way to work with files in Python is using a context manager with the `with` statement:

```python
with open('filename.txt', 'r') as file:
    # File operations here
    data = file.read()
    # Process data...
    if error_condition:
        return  # File still gets closed!

# File is automatically closed after the with block,
# even if an exception occurs or you return early
```

How It Works Behind the Scenes

When you use the `with` statement:

1. Python calls the `__enter__()` method of the file object (setup)
2. The block of code inside the `with` statement executes
3. When the block finishes (normally or due to an exception), Python calls the `__exit__()` method (teardown)

The `__exit__()` method ensures the file is closed properly.

Benefits:

- **Automatic cleanup**: Automatically closes the file, even if exceptions occur
- **Exception safety**: Resources are properly managed even if errors happen
- **Readability**: Cleaner, more readable code
- **Reduced bugs**: Prevents resource leaks by ensuring proper cleanup

Comparing C and Python Approaches:

C Approach	Python `with` Approach
Manual resource management	Automatic resource management
Must handle cleanup at every exit point	Single cleanup regardless of exit point
Easy to forget closing files	File always closed automatically
Need extra error checking	Error handling separated from resource management

Multiple Resources

You can even manage multiple files with a single `with` statement:

```python
with open('input.txt', 'r') as input_file, open('output.txt', 'w') as output_file:
    data = input_file.read()
    output_file.write(data.upper())
# Both files automatically closed
```

Closing Files Manually

If not using a context manager, you must close files explicitly:

```python
file = open('example.txt', 'r')
try:
    # File operations
    content = file.read()
except Exception as e:
    print(f"Error: {e}")
finally:
    file.close()  # Always executes, even if exception occurs
```

Why Closing Files is Critical:

1. **Resource Management**: Most operating systems limit the number of open files a process can have
2. **Data Integrity**: Some changes might be buffered and not written until the file is closed
3. **File Locking**: On some systems, other processes can't access files while they're open
4. **Performance**: Keeping files open unnecessarily can impact system performance

Checking if a File is Closed:

python

```python
file = open('example.txt', 'r')
print(file.closed)  # False
file.close()
print(file.closed)  # True
```

3. Reading from Files

Reading an Entire File

The simplest way to read a file is to load its entire contents into memory:

python

```python
with open('example.txt', 'r') as file:
    content = file.read()  # Reads the entire file into a single string
    print(content)
```

Size Limitations and Considerations:

python

```python
# Reading with size limit
with open('large_file.txt', 'r') as file:
    chunk = file.read(1024)  # Read only 1024 characters
    print(chunk)
```

Equivalent in C:

c

```c
FILE *file = fopen("example.txt", "r");
if (file == NULL) {
    perror("Error opening file");
    return -1;
}

// Get file size
fseek(file, 0, SEEK_END);
long file_size = ftell(file);
rewind(file);

// Allocate memory for entire file
char *buffer = (char *)malloc(file_size + 1);
if (buffer == NULL) {
    perror("Memory allocation failed");
    fclose(file);
    return -1;
}

// Read the file
size_t result = fread(buffer, 1, file_size, file);
if (result != file_size) {
    perror("Reading error");
    free(buffer);
    fclose(file);
    return -1;
}

// Null-terminate the string
buffer[file_size] = '\0';

// Use the content
printf("%s", buffer);

// Clean up
free(buffer);
fclose(file);
```

As you can see, Python's approach is much simpler and safer!

Reading Line by Line

For large files or when processing data line by line, Python offers several approaches:

Using readline()

```python
with open('example.txt', 'r') as file:
    line = file.readline()  # Read one line
    while line:
        print(line, end='')  # 'end' parameter prevents double newlines
        line = file.readline()
```

Using readlines()

```python
with open('example.txt', 'r') as file:
    lines = file.readlines()  # Returns a list of all lines
    for line in lines:
        print(line, end='')
```

Memory Considerations:

- `readlines()` loads all lines into memory at once (avoid for very large files)
- `readline()` reads one line at a time (more memory efficient)

Iterating Directly Over File Object (Best Practice)

```python
with open('example.txt', 'r') as file:
    for line in file:  # Most Pythonic way - file objects are iterable!
        print(line, end='')
```

This approach is:

- More memory efficient (reads one line at a time)
- More concise and readable
- Considered the "Pythonic" way to read files line by line

Understanding Line Endings

Python handles line endings differently depending on:

- The operating system
- The file open mode
- The `newline` parameter

```python
# Default behavior (converts to platform-specific newlines)
with open('example.txt', 'r') as file:
    lines = file.readlines()
    # On Windows: '\r\n' in the file becomes '\n' in memory
    # On Unix/Mac: '\n' stays as '\n'

# Preserve original newlines
with open('example.txt', 'r', newline='') as file:
    lines = file.readlines()
    # Newlines are preserved exactly as in the file
```

File Position and Seeking

The file object maintains a position pointer that advances as you read:

```python
with open('example.txt', 'r') as file:
    print(file.tell())       # 0 (beginning of file)
    data = file.read(5)      # Read 5 characters
    print(file.tell())       # 5 (moved forward)
    file.seek(0)             # Go back to beginning
    print(file.tell())       # 0 (reset to beginning)
    file.seek(10)            # Jump to position 10
    file.seek(0, 2)          # Go to end of file (2 is SEEK_END)
    end_pos = file.tell()    # Get file size
```

Seek Reference Points:

```python
# seek(offset, whence)
file.seek(0)         # Beginning of file (default whence=0)
file.seek(10, 0)     # 10 characters from beginning (SEEK_SET)
file.seek(10, 1)     # 10 characters from current position (SEEK_CUR)
file.seek(-10, 2)    # 10 characters before end of file (SEEK_END)
```

Comparing with C's fseek():

C Function	Python Equivalent	Description
`fseek(file, 0, SEEK_SET)`	`file.seek(0)`	Beginning of file
`fseek(file, 10, SEEK_SET)`	`file.seek(10)`	10 bytes from beginning
`fseek(file, 10, SEEK_CUR)`	`file.seek(10, 1)`	10 bytes from current position
`fseek(file, -10, SEEK_END)`	`file.seek(-10, 2)`	10 bytes before end of file

Working with CSV Files

CSV (Comma-Separated Values) files are common for structured data:

Basic CSV Reading Without the csv Module

```python
with open('data.csv', 'r') as file:
    for line in file:
        # Split by comma
        values = line.strip().split(',')
        print(values)
```

Problems with Simple Splitting:

- Doesn't handle commas within quoted fields
- Doesn't handle escape characters
- Can't properly parse complex CSV files

Using the csv Module (Recommended)

python

```python
import csv

# Reading CSV as lists
with open('data.csv', 'r', newline='') as csvfile:
    reader = csv.reader(csvfile)
    for row in reader:
        print(row)  # Each row is a list of values

# Reading CSV as dictionaries (first row as header)
with open('data.csv', 'r', newline='') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        print(row)  # Each row is a dictionary
        print(f"Name: {row['name']}, Age: {row['age']}")
```

CSV Dialect Options

python

```python
import csv

# Defining a custom dialect
csv.register_dialect('pipes', delimiter='|', quoting=csv.QUOTE_NONE)

with open('pipe_data.txt', 'r', newline='') as csvfile:
    reader = csv.reader(csvfile, dialect='pipes')
    for row in reader:
        print(row)
```

Advanced CSV Options

python

```python
import csv

with open('complex_data.csv', 'r', newline='') as csvfile:
    reader = csv.reader(
        csvfile,
        delimiter=',',          # Field separator
        quotechar='"',          # Quote character
        escapechar='\\',        # Escape character
        doublequote=True,       # "" represents a single quote in a field
        skipinitialspace=True   # Skip space after delimiter
    )
    for row in reader:
        print(row)
```

4. Writing to Files

Writing Text to Files

Basic Writing

python

```python
with open('output.txt', 'w') as file:
    file.write('Hello, World!\n')  # Returns number of characters written
    file.write('Python file handling is easy.\n')
```

Understanding the 'w' Mode:

- Creates the file if it doesn't exist
- Truncates (erases) the file if it exists
- Positions the file pointer at the beginning

Comparing with C's fprintf():

c

```c
FILE *file = fopen("output.txt", "w");
if (file == NULL) {
    perror("Error opening file");
    return -1;
}
fprintf(file, "Hello, World!\n");
fprintf(file, "C file handling requires more code.\n");
fclose(file);
```

Writing Multiple Lines

Using writelines()

python

```python
lines = ['First line\n', 'Second line\n', 'Third line\n']

with open('output.txt', 'w') as file:
    file.writelines(lines)  # Writes multiple strings at once
```

Note: `writelines()` doesn't add newlines automatically. Each string in the list should include a newline character if needed.

Alternative Approach with a Loop

python

```python
lines = ['First line', 'Second line', 'Third line']

with open('output.txt', 'w') as file:
    for line in lines:
        file.write(line + '\n')  # Explicitly add newline
```

Appending to Files

To add to a file without overwriting existing content:

python

```python
with open('log.txt', 'a') as file:  # 'a' for append mode
    file.write('New log entry\n')
```

Understanding the 'a' Mode:

- Creates the file if it doesn't exist
- Preserves existing content
- Positions the file pointer at the end of the file
- All writes will be added to the end

Real-world Example: Simple Logging

python

```python
def log_event(event, log_file='app.log'):
    from datetime import datetime
    timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

    with open(log_file, 'a') as file:
        file.write(f'[{timestamp}] {event}\n')

# Usage
log_event('Application started')
log_event('User logged in: alice')
log_event('Error: Database connection failed')
```

Flushing the Buffer

Python buffers file writes for performance. To force writing to disk:

python

```python
with open('important.txt', 'w') as file:
    file.write('Critical data')
    file.flush()  # Force write to disk
```

OS-level Syncing for More Reliability

python

```python
import os

with open('very_important.txt', 'w') as file:
    file.write('Extremely critical data')
    file.flush()
    os.fsync(file.fileno())  # Ensure it's physically written to disk
```

Writing Formatted Text

Using String Formatting

python

```python
data = [
    {'name': 'Alice', 'age': 30},
    {'name': 'Bob', 'age': 25},
    {'name': 'Charlie', 'age': 35}
]

with open('people.txt', 'w') as file:
    # Table header
    file.write(f"{'Name':<10}{'Age':<5}\n")
    file.write('-' * 15 + '\n')

    # Table data
    for person in data:
        file.write(f"{person['name']:<10}{person['age']:<5}\n")
```

Output:

```
Name      Age
---------------
Alice     30
Bob       25
Charlie   35
```

Using print() with file Parameter

python

```python
with open('formatted.txt', 'w') as file:
    print('Hello, World!', file=file)
    print('This is printed to a file.', file=file)

    # With formatting
    for i in range(1, 6):
        print(f"{i} squared is {i**2}", file=file)
```

Writing to CSV Files

python

```python
import csv

data = [
    ['Name', 'Age', 'Country'],
    ['Alice', 25, 'USA'],
    ['Bob', 30, 'Canada'],
    ['Charlie', 35, 'UK']
]

with open('people.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile)
    writer.writerows(data)  # Write all rows at once
```

Writing Dictionaries to CSV

python

```python
import csv

data = [
    {'Name': 'Alice', 'Age': 25, 'Country': 'USA'},
    {'Name': 'Bob', 'Age': 30, 'Country': 'Canada'},
    {'Name': 'Charlie', 'Age': 35, 'Country': 'UK'}
]

with open('people_dict.csv', 'w', newline='') as csvfile:
    fieldnames = ['Name', 'Age', 'Country']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()  # Write header row
    writer.writerows(data)  # Write all data rows
```

CSV Writing Options

python

```python
import csv

data = [['Id,1', 'Name with, comma'], ['Id,2', 'Another, name']]

with open('escaped.csv', 'w', newline='') as csvfile:
    writer = csv.writer(
        csvfile,
        delimiter=',',
        quotechar='"',
        quoting=csv.QUOTE_ALL  # Quote all fields
    )
    writer.writerows(data)
```

5. Working with Binary Files

Understanding Binary Mode

Binary mode preserves the exact byte values without any transformation:

python

```python
# Writing binary data
with open('data.bin', 'wb') as file:  # Note 'b' in mode
    file.write(b'\x00\x01\x02\x03')  # Bytes literal
```

Key Differences from Text Mode:

- No encoding/decoding of text
- No newline translation
- Data is read/written as bytes objects, not strings
- Perfect for non-text files (images, audio, video, etc.)

Reading and Writing Binary Data

python

```python
# Writing binary data
with open('binary_file.bin', 'wb') as file:
    # Bytes literals start with b
    file.write(b'\x48\x65\x6C\x6C\x6F')  # "Hello" in ASCII/bytes

# Reading binary data
with open('binary_file.bin', 'rb') as file:
    data = file.read()
    print(data)  # b'\x48\x65\x6C\x6C\x6F'
    print(len(data))  # 5 bytes
```

Converting Between Strings and Bytes

python

```python
# String to bytes
text = "Hello, World!"
binary_data = text.encode('utf-8')  # Convert to bytes with specific encoding

with open('encoded.bin', 'wb') as file:
    file.write(binary_data)

# Bytes to string
with open('encoded.bin', 'rb') as file:
    binary_data = file.read()
    text = binary_data.decode('utf-8')  # Convert back to string
    print(text)  # "Hello, World!"
```

Working with Images and Other Binary Files

python

```python
# Copy an image file
with open('original.jpg', 'rb') as source:
    with open('copy.jpg', 'wb') as destination:
        # Read in chunks to handle large files
        chunk_size = 4096
        while True:
            chunk = source.read(chunk_size)
            if not chunk:  # End of file
                break
            destination.write(chunk)
```

Using the Pickle Module

Pickle allows you to serialize Python objects to binary files:

python

```python
import pickle

# Data to serialize
data = {
    'name': 'Alice',
    'age': 25,
    'grades': [90, 85, 95],
    'active': True,
    'address': {
        'street': '123 Main St',
        'city': 'Anytown'
    }
}

# Saving Python objects to a file
with open('data.pkl', 'wb') as file:
    pickle.dump(data, file)

# Loading Python objects from a file
with open('data.pkl', 'rb') as file:
    loaded_data = pickle.load(file)
    print(loaded_data)  # Same as original data
    print(loaded_data['grades'][0])  # 90
```

Security Warning

Pickle is not secure against maliciously constructed data. Never unpickle data from untrusted sources!

Using Structured Binary Data with `struct`

For precise control over binary formats:

```python
import struct

# Pack values into binary format
# Format: 'I' (unsigned int), 'f' (float), '10s' (10-char string)
values = (123456, 3.14, b'Hello')
packed_data = struct.pack('I f 10s', *values)

with open('struct_data.bin', 'wb') as file:
    file.write(packed_data)

# Unpack binary data
with open('struct_data.bin', 'rb') as file:
    binary_data = file.read()
    unpacked_data = struct.unpack('I f 10s', binary_data)
    print(unpacked_data)  # (123456, 3.14, b'Hello\x00\x00\x00\x00\x00')
```

6. Error Handling and Exception Management

Common File Exceptions

When working with files, several exceptions might occur:

Exception	Description
`FileNotFoundError`	The file or directory does not exist
`PermissionError`	You lack permission to access the file
`IsADirectoryError`	You're trying to open a directory as a file
`FileExistsError`	You're trying to create a file that already exists (with certain modes)
`OSError`	General operating system error related to files
`IOError`	General input/output error (base class for many file errors)
`UnicodeDecodeError`	Error when decoding the file with the wrong encoding

Handling File Existence

Always check if a file exists before attempting operations:

```python
import os

filename = 'data.txt'

# Method 1: Check before opening
if os.path.exists(filename):
    with open(filename, 'r') as file:
        content = file.read()
else:
    print(f"The file '{filename}' does not exist")
    # Optionally create the file
    with open(filename, 'w') as file:
        file.write("Initial content\n")
```

Try-Except Pattern for File Operations

The most robust approach is using try-except blocks:

python

```python
try:
    with open('file_that_might_not_exist.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("The file does not exist")
    # You could create the file or use a default value
    content = "Default content since file was not found"
except PermissionError:
    print("You don't have permission to access this file")
except IsADirectoryError:
    print("The specified path is a directory, not a file")
except UnicodeDecodeError:
    print("Could not decode the file with the current encoding")
    # Try with a different encoding
    try:
        with open('file_that_might_not_exist.txt', 'r', encoding='latin-1') as file:
            content = file.read()
    except Exception as e:
        print(f"Still couldn't read the file: {e}")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

Creating Files Safely

When creating files, handle potential issues:

python

```python
try:
    with open('output.txt', 'x') as file:  # 'x' mode fails if file exists
        file.write("New file content")
except FileExistsError:
    print("The file already exists")
    # Ask user if they want to overwrite
    overwrite = input("Do you want to overwrite? (y/n): ")
    if overwrite.lower() == 'y':
        with open('output.txt', 'w') as file:
            file.write("Overwritten content")
```

Recovery Strategies

Implement recovery strategies when file operations fail:

python

```python
def safe_read_file(filename, default_value=None):
    """Safely read a file with fallback to default value."""
    try:
        with open(filename, 'r') as file:
            return file.read()
    except FileNotFoundError:
        print(f"Warning: File '{filename}' not found, using default value")
        return default_value
    except Exception as e:
        print(f"Error reading file '{filename}': {e}")
        return default_value

# Usage
config_data = safe_read_file('config.txt', default_value="{}")
```

File Locking

For multi-process applications, implement file locking:

python

```python
import fcntl  # Unix-based systems only

def locked_write(filename, data):
    """Write to a file with an exclusive lock."""
    with open(filename, 'w') as file:
        try:
            # Get an exclusive lock
            fcntl.flock(file, fcntl.LOCK_EX | fcntl.LOCK_NB)

            # Write data
            file.write(data)

            # Release the lock
            fcntl.flock(file, fcntl.LOCK_UN)
            return True
        except IOError:
            print("File is locked by another process")
            return False
```

Handling Path Issues

Deal with path-related problems:

```python
import os

def ensure_directory_exists(filepath):
    """Ensure the directory for the file exists."""
    directory = os.path.dirname(filepath)

    # If directory is empty, it means the current directory
    if directory and not os.path.exists(directory):
        try:
            os.makedirs(directory)
            print(f"Created directory: {directory}")
        except PermissionError:
            print(f"No permission to create directory: {directory}")
            return False
        except FileExistsError:
            # This can happen with race conditions
            if not os.path.isdir(directory):
                print(f"Path exists but is not a directory: {directory}")
                return False

    return True

# Usage
filepath = "data/logs/app.log"
if ensure_directory_exists(filepath):
    with open(filepath, 'a') as file:
        file.write("Log entry\n")
```

7. Practical Examples

Let's explore some real-world applications of file handling in Python.

Example 1: Creating a Simple Log System

This example demonstrates a complete logging system with timestamps:

python

```python
import datetime
import os

class SimpleLogger:
    def __init__(self, log_file='app.log'):
        self.log_file = log_file
        # Ensure the log directory exists
        log_dir = os.path.dirname(log_file)
        if log_dir and not os.path.exists(log_dir):
            os.makedirs(log_dir)

    def log(self, message, level='INFO'):
        """Log a message with timestamp and level."""
        timestamp = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        log_entry = f'[{timestamp}] [{level}] {message}\n'

        try:
            with open(self.log_file, 'a') as file:
                file.write(log_entry)
        except Exception as e:
            print(f"Error writing to log: {e}")

    def info(self, message):
        """Log an info message."""
        self.log(message, 'INFO')

    def warning(self, message):
        """Log a warning message."""
        self.log(message, 'WARNING')

    def error(self, message):
        """Log an error message."""
        self.log(message, 'ERROR')

# Usage
logger = SimpleLogger('logs/application.log')
logger.info('Application started')

try:
    # Simulate an error
    result = 10 / 0
except Exception as e:
    logger.error(f'Division error: {e}')
```

```
logger.info('Operation completed')
```

This logging system:

- Creates directories if needed
- Adds timestamps and log levels
- Provides different logging methods
- Handles exceptions gracefully

Example 2: Configuration File Manager

This example implements a complete configuration system:

python

```python
import os
import json

class ConfigManager:
    def __init__(self, config_file='config.json'):
        self.config_file = config_file
        self.config = {}
        self.load()

    def load(self):
        """Load configuration from file."""
        try:
            if os.path.exists(self.config_file):
                with open(self.config_file, 'r') as file:
                    self.config = json.load(file)
            else:
                # Create default config
                self.config = {
                    'app_name': 'MyApp',
                    'version': '1.0',
                    'debug': False,
                    'log_level': 'INFO',
                    'data_dir': 'data'
                }
                self.save()  # Save default config
        except Exception as e:
            print(f"Error loading config: {e}")
            # Use default values if loading fails
            self.config = {
                'app_name': 'MyApp',
                'version': '1.0',
                'debug': True,  # Debug mode by default on error
                'log_level': 'DEBUG',
                'data_dir': 'data'
            }

    def save(self):
        """Save configuration to file."""
        try:
            # Ensure directory exists
            config_dir = os.path.dirname(self.config_file)
            if config_dir and not os.path.exists(config_dir):
                os.makedirs(config_dir)
```

```
            with open(self.config_file, 'w') as file:
                json.dump(self.config, file, indent=4)
            return True
        except Exception as e:
            print(f"Error saving config: {e}")
            return False

    def get(self, key, default=None):
        """Get a configuration value."""
        return self.config.get(key, default)

    def set(self, key, value):
        """Set a configuration value and save."""
        self.config[key] = value
        return self.save()

# Usage
config = ConfigManager('app_config.json')
print(f"App name: {config.get('app_name')}")
print(f"Debug mode: {config.get('debug')}")

# Update a setting
config.set('debug', True)
```

This configuration manager:

- Loads/saves JSON configuration
- Provides default values
- Handles missing files
- Offers simple get/set interface
- Creates directories automatically

Example 3: CSV Data Processor

This example demonstrates complete CSV processing:

python

```
import csv
import os
from datetime import datetime

class CSVProcessor:
    def __init__(self, input_file):
        self.input_file = input_file
        self.data = []
        self.headers = []

    def load(self):
        """Load data from CSV file."""
        try:
            with open(self.input_file, 'r', newline='') as file:
                reader = csv.reader(file)
                self.headers = next(reader)  # First row as headers
                self.data = list(reader)     # Rest of the data
            return True
        except FileNotFoundError:
            print(f"File not found: {self.input_file}")
            return False
        except Exception as e:
            print(f"Error loading CSV: {e}")
            return False

    def save(self, output_file):
        """Save data to a new CSV file."""
        try:
            # Ensure directory exists
            output_dir = os.path.dirname(output_file)
            if output_dir and not os.path.exists(output_dir):
                os.makedirs(output_dir)

            with open(output_file, 'w', newline='') as file:
                writer = csv.writer(file)
                writer.writerow(self.headers)  # Write headers
                writer.writerows(self.data)    # Write data
            return True
        except Exception as e:
            print(f"Error saving CSV: {e}")
            return False

    def add_row(self, row):
```

```
        """Add a new row of data."""
        if len(row) != len(self.headers):
            print("Row length doesn't match headers")
            return False
        self.data.append(row)
        return True

    def get_column(self, column_name):
        """Get all values for a specific column."""
        if column_name not in self.headers:
            print(f"Column not found: {column_name}")
            return []

        column_index = self.headers.index(column_name)
        return [row[column_index] for row in self.data]

    def filter_rows(self, column_name, value):
        """Filter rows where column matches value."""
        if column_name not in self.headers:
            print(f"Column not found: {column_name}")
            return []

        column_index = self.headers.index(column_name)
        return [row for row in self.data if row[column_index] == value]

    def calculate_average(self, column_name):
        """Calculate average for a numeric column."""
        values = self.get_column(column_name)
        try:
            numeric_values = [float(val) for val in values if val]
            if not numeric_values:
                return 0
            return sum(numeric_values) / len(numeric_values)
        except ValueError:
            print(f"Column contains non-numeric values: {column_name}")
            return None

# Usage example
processor = CSVProcessor('data/sales.csv')
if processor.load():
    # Analyze data
    total_sales = processor.calculate_average('Amount')
    print(f"Average sale amount: ${total_sales:.2f}")
```

```
    # Filter data
    january_sales = processor.filter_rows('Month', 'January')
    print(f"January sales count: {len(january_sales)}")

    # Add a new record
    today = datetime.now().strftime('%Y-%m-%d')
    processor.add_row(['2023', 'December', 'Online', '150.75', today])

    # Save modified data
    processor.save('data/updated_sales.csv')
```

This CSV processor:

- Loads and saves CSV files
- Extracts columns and filters rows
- Performs calculations on numeric data
- Handles errors gracefully
- Maintains data structure integrity

Example 4: Simple File Encryption/Decryption

This example implements basic file encryption:

python

```python
import os
from cryptography.fernet import Fernet

class SimpleFileEncryptor:
    def __init__(self, key_file='encryption_key.key'):
        self.key_file = key_file
        self.key = self._load_or_generate_key()
        self.cipher = Fernet(self.key)

    def _load_or_generate_key(self):
        """Load existing key or generate a new one."""
        try:
            if os.path.exists(self.key_file):
                with open(self.key_file, 'rb') as file:
                    return file.read()
            else:
                # Generate new key
                key = Fernet.generate_key()
                with open(self.key_file, 'wb') as file:
                    file.write(key)
                return key
        except Exception as e:
            print(f"Error with encryption key: {e}")
            # Generate temporary key (won't be saved)
            return Fernet.generate_key()

    def encrypt_file(self, input_file, output_file=None):
        """Encrypt a file."""
        if not output_file:
            output_file = input_file + '.encrypted'

        try:
            with open(input_file, 'rb') as file:
                data = file.read()

            # Encrypt the data
            encrypted_data = self.cipher.encrypt(data)

            with open(output_file, 'wb') as file:
                file.write(encrypted_data)

            return True
        except FileNotFoundError:
```

```
            print(f"File not found: {input_file}")
            return False
        except Exception as e:
            print(f"Encryption error: {e}")
            return False

    def decrypt_file(self, input_file, output_file=None):
        """Decrypt a file."""
        if not output_file:
            if input_file.endswith('.encrypted'):
                output_file = input_file[:-10]  # Remove .encrypted
            else:
                output_file = input_file + '.decrypted'

        try:
            with open(input_file, 'rb') as file:
                encrypted_data = file.read()

            # Decrypt the data
            decrypted_data = self.cipher.decrypt(encrypted_data)

            with open(output_file, 'wb') as file:
                file.write(decrypted_data)

            return True
        except FileNotFoundError:
            print(f"File not found: {input_file}")
            return False
        except Exception as e:
            print(f"Decryption error: {e}")
            return False

# Usage
encryptor = SimpleFileEncryptor()

# Encrypt a file
if encryptor.encrypt_file('sensitive_data.txt'):
    print("File encrypted successfully")

# Decrypt a file
if encryptor.decrypt_file('sensitive_data.txt.encrypted'):
    print("File decrypted successfully")
```

This file encryptor:

- Generates and saves encryption keys
- Encrypts files with strong encryption
- Decrypts previously encrypted files
- Handles file paths and errors

8. Best Practices for File Handling

1. Always Use Context Managers

Use the `with` statement whenever possible:

```python
# GOOD: Files automatically closed
with open('file.txt', 'r') as file:
    data = file.read()

# BAD: Manual closing required, risk of resource leaks
file = open('file.txt', 'r')
data = file.read()
file.close()  # Might never execute if exception occurs
```

2. Handle Exceptions Properly

Always implement error handling for file operations:

python

```python
# GOOD: Comprehensive error handling
try:
    with open('file.txt', 'r') as file:
        data = file.read()
except FileNotFoundError:
    print("File not found, creating with default content")
    with open('file.txt', 'w') as file:
        file.write("Default content")
except PermissionError:
    print("Permission denied")
except Exception as e:
    print(f"Unexpected error: {e}")

# BAD: No error handling
with open('file.txt', 'r') as file:  # Will crash if file doesn't exist
    data = file.read()
```

3. Use Appropriate File Modes

Choose the correct mode based on your needs:

python

```python
# GOOD: Using appropriate modes
with open('file.txt', 'r') as file:  # Read-only
    data = file.read()

with open('new_file.txt', 'w') as file:  # Write (truncate existing)
    file.write("New content")

with open('append_file.txt', 'a') as file:  # Append
    file.write("Additional content")

# BAD: Using 'w' when 'a' is intended
with open('log.txt', 'w') as file:  # Erases all previous logs!
    file.write("New log entry")
```

4. Be Mindful of File Sizes

For large files, read in chunks instead of loading everything at once:

python

```python
# GOOD: Processing large files in chunks
with open('huge_file.txt', 'r') as file:
    while True:
        chunk = file.read(4096)  # 4KB chunks
        if not chunk:
            break
        process_data(chunk)

# BAD: Loading entire large file
with open('huge_file.txt', 'r') as file:
    data = file.read()  # May consume too much memory
    process_data(data)
```

5. Validate File Paths

Check if files exist before operations:

python

```python
# GOOD: Check before operation
import os

file_path = 'data/report.txt'
if os.path.exists(file_path):
    with open(file_path, 'r') as file:
        data = file.read()
else:
    print(f"Warning: {file_path} doesn't exist")

# BETTER: Use try-except
try:
    with open(file_path, 'r') as file:
        data = file.read()
except FileNotFoundError:
    print(f"Warning: {file_path} doesn't exist")
```

6. Back Up Important Files Before Writing

For critical data, create backups before modifying:

python

```python
import shutil
import os

file_path = 'important_config.txt'

# Create backup if file exists
if os.path.exists(file_path):
    backup_path = file_path + '.bak'
    shutil.copy2(file_path, backup_path)
    print(f"Backup created: {backup_path}")

# Now safe to modify
with open(file_path, 'w') as file:
    file.write("New configuration")
```

7. Use Specialized Libraries

Use appropriate libraries for specific file formats:

python

```python
# CSV files
import csv
with open('data.csv', 'r', newline='') as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row)

# JSON files
import json
with open('config.json', 'r') as file:
    config = json.load(file)

# YAML files
import yaml
with open('config.yaml', 'r') as file:
    config = yaml.safe_load(file)

# Excel files
import pandas as pd
data = pd.read_excel('data.xlsx')
```

8. Check File Existence Before Reading or Writing

```python
import os

# For reading: check before opening
file_path = 'data.txt'
if os.path.isfile(file_path):  # Specifically a file, not directory
    with open(file_path, 'r') as file:
        content = file.read()
else:
    print(f"File doesn't exist: {file_path}")

# For writing: create directories if needed
output_path = 'reports/monthly/summary.txt'
os.makedirs(os.path.dirname(output_path), exist_ok=True)
with open(output_path, 'w') as file:
    file.write("Monthly report")
```

9. Use Absolute Paths When Necessary

For reliability in complex applications:

```python
import os

# Get current script directory
script_dir = os.path.dirname(os.path.abspath(__file__))

# Construct absolute paths
config_path = os.path.join(script_dir, 'config', 'settings.ini')
data_path = os.path.join(script_dir, 'data', 'records.csv')

# Now these paths work regardless of current working directory
with open(config_path, 'r') as file:
    settings = file.read()
```

10. Add Proper Documentation

Document file operations clearly:

python

```python
def save_user_data(user_id, data, filepath=None):
    """
    Save user data to a JSON file.

    Args:
        user_id (int): The unique identifier for the user
        data (dict): User data to save
        filepath (str, optional): Path to save the file.
                                  If None, uses default location.

    Returns:
        bool: True if save was successful, False otherwise

    Raises:
        ValueError: If user_id is not a positive integer
        TypeError: If data is not a dictionary
    """
    if not isinstance(user_id, int) or user_id <= 0:
        raise ValueError("user_id must be a positive integer")

    if not isinstance(data, dict):
        raise TypeError("data must be a dictionary")

    if filepath is None:
        filepath = f"users/{user_id}.json"

    # Create directory if it doesn't exist
    os.makedirs(os.path.dirname(filepath), exist_ok=True)

    try:
        with open(filepath, 'w') as file:
            json.dump(data, file, indent=2)
        return True
    except Exception as e:
        print(f"Error saving user data: {e}")
        return False
```

9. Practice Exercises

Exercise 1: Word Counter

Write a program that reads a text file and counts:

- Total number of lines
- Total number of words
- Total number of characters

python

```python
def analyze_text_file(filepath):
    """
    Analyze a text file and count lines, words, and characters.

    Args:
        filepath (str): Path to the text file

    Returns:
        dict: Dictionary with count information
    """
    try:
        with open(filepath, 'r', encoding='utf-8') as file:
            content = file.read()
            lines = content.split('\n')
            words = content.split()
            chars = len(content)

            return {
                'lines': len(lines),
                'words': len(words),
                'characters': chars
            }
    except FileNotFoundError:
        print(f"File not found: {filepath}")
        return None
    except Exception as e:
        print(f"Error analyzing file: {e}")
        return None

# Usage
result = analyze_text_file('sample.txt')
if result:
    print(f"Lines: {result['lines']}")
    print(f"Words: {result['words']}")
    print(f"Characters: {result['characters']}")
```

Exercise 2: File Merger

Create a program that takes two text files as input and creates a new file containing the merged contents of both files, alternating lines from each file:

python

```python
def merge_files(file1_path, file2_path, output_path):
    """
    Merge two text files, alternating lines from each.

    Args:
        file1_path (str): Path to first input file
        file2_path (str): Path to second input file
        output_path (str): Path to output file

    Returns:
        bool: True if successful, False otherwise
    """
    try:
        # Read both files
        with open(file1_path, 'r') as file1:
            lines1 = file1.readlines()

        with open(file2_path, 'r') as file2:
            lines2 = file2.readlines()

        # Determine the maximum number of lines
        max_lines = max(len(lines1), len(lines2))

        # Merge files line by line
        with open(output_path, 'w') as output:
            for i in range(max_lines):
                # Add line from first file if available
                if i < len(lines1):
                    output.write(lines1[i].rstrip('\n') + '\n')

                # Add line from second file if available
                if i < len(lines2):
                    output.write(lines2[i].rstrip('\n') + '\n')

        return True
    except Exception as e:
        print(f"Error merging files: {e}")
        return False

# Usage
success = merge_files('file1.txt', 'file2.txt', 'merged.txt')
```

```
if success:
    print("Files merged successfully!")
```

Exercise 3: Student Grade Analyzer

Write a script that reads a CSV file containing student grades and:

- Calculates the average grade for each student
- Finds the highest and lowest scoring students
- Outputs the results to a new CSV file

python

```
import csv

def analyze_grades(input_csv, output_csv):
    """
    Analyze student grades from a CSV file.

    Args:
        input_csv (str): Path to input CSV file
        output_csv (str): Path to output CSV file

    Returns:
        dict: Summary statistics
    """
    try:
        students = {}

        # Read input CSV
        with open(input_csv, 'r', newline='') as file:
            reader = csv.DictReader(file)
            for row in reader:
                student_name = row['Student']
                grades = []

                # Extract all grade columns
                for key, value in row.items():
                    if key != 'Student' and value.strip():
                        try:
                            grade = float(value)
                            grades.append(grade)
                        except ValueError:
                            print(f"Warning: Invalid grade '{value}' for {student_name}")

                # Calculate average if there are grades
                if grades:
                    avg = sum(grades) / len(grades)
                    students[student_name] = avg

        # Find highest and lowest students
        if students:
            highest_student = max(students.items(), key=lambda x: x[1])
            lowest_student = min(students.items(), key=lambda x: x[1])

            # Write results to output CSV
```

```
            with open(output_csv, 'w', newline='') as file:
                writer = csv.writer(file)
                writer.writerow(['Student', 'Average Grade'])

                # Sort by average grade (highest first)
                for student, avg in sorted(students.items(), key=lambda x: x[1], reverse=True):
                    writer.writerow([student, f"{avg:.2f}"])

                # Add summary
                writer.writerow([])
                writer.writerow(['Summary', ''])
                writer.writerow(['Highest', f"{highest_student[0]} ({highest_student[1]:.2f})"]
                writer.writerow(['Lowest', f"{lowest_student[0]} ({lowest_student[1]:.2f})"])
                writer.writerow(['Class Average', f"{sum(students.values()) / len(students):.2f

            return {
                'total_students': len(students),
                'highest': highest_student,
                'lowest': lowest_student,
                'class_average': sum(students.values()) / len(students)
            }
        else:
            print("No valid student data found")
            return None
    except Exception as e:
        print(f"Error analyzing grades: {e}")
        return None

# Usage
results = analyze_grades('grades.csv', 'grade_analysis.csv')
if results:
    print(f"Analyzed {results['total_students']} students")
    print(f"Highest: {results['highest'][0]} ({results['highest'][1]:.2f})")
    print(f"Lowest: {results['lowest'][0]} ({results['lowest'][1]:.2f})")
    print(f"Class Average: {results['class_average']:.2f}")
```

Exercise 4: Simple File Encryption

Create a simple "file encryption" program that:

- Reads a text file
- Performs a Caesar cipher encryption (shift each character)

- Writes the encrypted text to a new file
- Can also decrypt files it has encrypted

python

```
def caesar_cipher(text, shift, decrypt=False):
    """
    Apply Caesar cipher to text.

    Args:
        text (str): Text to encrypt/decrypt
        shift (int): Shift value (1-25)
        decrypt (bool): True for decryption, False for encryption

    Returns:
        str: Encrypted/decrypted text
    """
    # Ensure shift is in valid range
    shift = shift % 26

    # Reverse shift for decryption
    if decrypt:
        shift = -shift

    result = ""

    for char in text:
        if char.isalpha():
            # Determine ASCII offset (65 for uppercase, 97 for lowercase)
            ascii_offset = 65 if char.isupper() else 97

            # Apply Caesar cipher formula: (x + shift) % 26
            shifted = (ord(char) - ascii_offset + shift) % 26

            # Convert back to character
            result += chr(shifted + ascii_offset)
        else:
            # Non-alphabetic characters remain unchanged
            result += char

    return result

def encrypt_file(input_path, output_path, shift):
    """
    Encrypt a file using Caesar cipher.

    Args:
        input_path (str): Path to input file
```

```
        output_path (str): Path to output file
        shift (int): Shift value for encryption

    Returns:
        bool: True if successful, False otherwise
    """
    try:
        with open(input_path, 'r') as input_file:
            content = input_file.read()

        # Encrypt the content
        encrypted = caesar_cipher(content, shift)

        with open(output_path, 'w') as output_file:
            # Save the shift value at the beginning (for decryption)
            output_file.write(f"#SHIFT:{shift}#\n")
            output_file.write(encrypted)

        return True
    except Exception as e:
        print(f"Error encrypting file: {e}")
        return False

def decrypt_file(input_path, output_path):
    """
    Decrypt a file encrypted with this program.

    Args:
        input_path (str): Path to encrypted file
        output_path (str): Path for decrypted output

    Returns:
        bool: True if successful, False otherwise
    """
    try:
        with open(input_path, 'r') as input_file:
            content = input_file.read()

        # Extract the shift value from the header
        if content.startswith("#SHIFT:"):
            header_end = content.find("#\n")
            if header_end > 7:  # "#SHIFT:" is 7 characters
                shift_str = content[7:header_end]
                try:
```

```
                    shift = int(shift_str)
                    content = content[header_end + 2:]  # Skip header

                    # Decrypt the content
                    decrypted = caesar_cipher(content, shift, decrypt=True)

                    with open(output_path, 'w') as output_file:
                        output_file.write(decrypted)

                    return True
                except ValueError:
                    print("Invalid shift value in file header")
            else:
                print("Invalid file header format")
        else:
            print("This doesn't appear to be an encrypted file (no shift header)")

        return False
    except Exception as e:
        print(f"Error decrypting file: {e}")
        return False

# Usage
# Encrypt a file
if encrypt_file('secret.txt', 'secret.encrypted', 7):
    print("File encrypted successfully")

# Decrypt a file
if decrypt_file('secret.encrypted', 'secret.decrypted'):
    print("File decrypted successfully")
```

10. Advanced Topics in File Handling

Working with Binary Data in Structured Formats

For applications that need to work with binary data in a structured way:

python

```
import struct

# Define a record structure for a binary file
# Format: "IHd" (unsigned int, unsigned short, double)
record_format = "IHd"
record_size = struct.calcsize(record_format)

# Create structured binary data
def write_binary_records(filename, records):
    """Write structured records to a binary file."""
    with open(filename, 'wb') as file:
        # Write the number of records as header
        file.write(struct.pack("I", len(records)))

        # Write each record
        for record in records:
            id_num, age, score = record
            file.write(struct.pack(record_format, id_num, age, score))

# Read structured binary data
def read_binary_records(filename):
    """Read structured records from a binary file."""
    records = []

    with open(filename, 'rb') as file:
        # Read the number of records
        header_data = file.read(4)  # 4 bytes for unsigned int
        num_records = struct.unpack("I", header_data)[0]

        # Read each record
        for _ in range(num_records):
            record_data = file.read(record_size)
            if not record_data or len(record_data) < record_size:
                break  # End of file or incomplete record

            id_num, age, score = struct.unpack(record_format, record_data)
            records.append((id_num, age, score))

    return records

# Usage example
records = [
    (1001, 25, 92.5),
```

```
    (1002, 19, 88.0),
    (1003, 32, 96.7)
]
write_binary_records('student_records.bin', records)
read_records = read_binary_records('student_records.bin')
for record in read_records:
    print(f"ID: {record[0]}, Age: {record[1]}, Score: {record[2]}")
```

Memory-Mapped Files

For very large files that need frequent access:

python

```python
import mmap
import os

def search_large_file(filename, term):
    """Search a large file efficiently using memory mapping."""
    found_lines = []

    with open(filename, 'r+b') as file:
        # Get file size
        file_size = os.path.getsize(filename)
        if file_size == 0:
            return found_lines

        # Create memory-mapped file object
        with mmap.mmap(file.fileno(), 0) as mmapped_file:
            # Search for term
            line_num = 0
            mmapped_file.seek(0)

            for line in iter(mmapped_file.readline, b''):
                line_num += 1
                if term.encode() in line:
                    found_lines.append((line_num, line.decode().strip()))

    return found_lines

# Usage
results = search_large_file('huge_log_file.txt', 'ERROR')
for line_num, line_text in results:
    print(f"Line {line_num}: {line_text}")
```

Handling File Paths Across Platforms

To ensure cross-platform compatibility:

python

```python
import os
from pathlib import Path

# Old style (OS-specific)
windows_path = 'C:\\Users\\username\\Documents\\file.txt'
unix_path = '/home/username/documents/file.txt'

# Using os.path (better, handles platform differences)
data_dir = os.path.join('data', 'processed')
config_path = os.path.join('config', 'settings.ini')

# Using pathlib (modern, object-oriented approach)
data_dir = Path('data') / 'processed'
config_path = Path('config') / 'settings.ini'

# Getting absolute paths
abs_path = Path('relative/path').absolute()
home_dir = Path.home()
user_docs = home_dir / 'Documents'

# Platform-specific paths
if os.name == 'nt':  # Windows
    app_data = Path(os.environ['APPDATA']) / 'MyApp'
else:  # Unix/Linux/Mac
    app_data = Path.home() / '.config' / 'myapp'

# Creating paths
app_data.mkdir(parents=True, exist_ok=True)

# Checking paths
if user_docs.exists() and user_docs.is_dir():
    for file_path in user_docs.glob('*.txt'):
        print(file_path.name)
```

File Change Monitoring

For applications that need to watch file changes:

python

```
import time
import os
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler

class FileChangeHandler(FileSystemEventHandler):
    def on_created(self, event):
        if not event.is_directory:
            print(f"File created: {event.src_path}")
            process_new_file(event.src_path)

    def on_modified(self, event):
        if not event.is_directory:
            print(f"File modified: {event.src_path}")
            process_updated_file(event.src_path)

    def on_deleted(self, event):
        if not event.is_directory:
            print(f"File deleted: {event.src_path}")
            handle_deleted_file(event.src_path)

def process_new_file(path):
    """Process a newly created file."""
    print(f"Processing new file: {path}")
    # Implementation here

def process_updated_file(path):
    """Process an updated file."""
    print(f"Processing updated file: {path}")
    # Implementation here

def handle_deleted_file(path):
    """Handle a deleted file."""
    print(f"Handling deleted file: {path}")
    # Implementation here

def watch_directory(path):
    """Watch a directory for changes."""
    event_handler = FileChangeHandler()
    observer = Observer()
    observer.schedule(event_handler, path, recursive=True)
    observer.start()
```

```
    try:
        print(f"Watching directory: {path}")
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        observer.stop()

    observer.join()

# Usage
watch_directory('data')
```

Working with Compressed Files

Handle ZIP, GZIP, and other compressed formats:

python

```python
import zipfile
import gzip
import tarfile
import os

# ZIP files
def create_zip_archive(directory, output_zip):
    """Create a ZIP archive from a directory."""
    with zipfile.ZipFile(output_zip, 'w', zipfile.ZIP_DEFLATED) as zipf:
        for root, _, files in os.walk(directory):
            for file in files:
                file_path = os.path.join(root, file)
                arcname = os.path.relpath(file_path, directory)
                zipf.write(file_path, arcname)

def extract_zip_archive(zip_file, output_dir):
    """Extract a ZIP archive to a directory."""
    with zipfile.ZipFile(zip_file, 'r') as zipf:
        zipf.extractall(output_dir)

def list_zip_contents(zip_file):
    """List contents of a ZIP file."""
    with zipfile.ZipFile(zip_file, 'r') as zipf:
        return zipf.namelist()

# GZIP files (single file compression)
def compress_with_gzip(input_file, output_file=None):
    """Compress a file using GZIP."""
    if output_file is None:
        output_file = input_file + '.gz'

    with open(input_file, 'rb') as f_in:
        with gzip.open(output_file, 'wb') as f_out:
            f_out.writelines(f_in)

    return output_file

def decompress_gzip(gzip_file, output_file=None):
    """Decompress a GZIP file."""
    if output_file is None:
        output_file = os.path.splitext(gzip_file)[0]

    with gzip.open(gzip_file, 'rb') as f_in:
```

```
        with open(output_file, 'wb') as f_out:
            f_out.write(f_in.read())

    return output_file

# TAR archives (with or without compression)
def create_tar_archive(directory, output_tar, compression=None):
    """Create a TAR archive from a directory."""
    mode = 'w'
    if compression == 'gzip':
        mode = 'w:gz'
    elif compression == 'bzip2':
        mode = 'w:bz2'
    elif compression == 'xz':
        mode = 'w:xz'

    with tarfile.open(output_tar, mode) as tar:
        tar.add(directory, arcname=os.path.basename(directory))

def extract_tar_archive(tar_file, output_dir):
    """Extract a TAR archive to a directory."""
    with tarfile.open(tar_file, 'r:*') as tar:
        tar.extractall(path=output_dir)

# Usage examples
create_zip_archive('project', 'project.zip')
files = list_zip_contents('project.zip')
extract_zip_archive('project.zip', 'extracted')

compress_with_gzip('large_file.txt')
decompress_gzip('large_file.txt.gz')

create_tar_archive('project', 'project.tar.gz', compression='gzip')
extract_tar_archive('project.tar.gz', 'extracted')
```

Temporary Files

For operations requiring temporary files:

python

```python
import tempfile
import os
import shutil

# Create a temporary file
with tempfile.NamedTemporaryFile(delete=False) as temp:
    temp_name = temp.name
    temp.write(b'Temporary data\n')
    # More operations...

# The file still exists after the context manager
print(f"Temporary file created: {temp_name}")
os.unlink(temp_name)  # Manually delete when done

# Create a temporary file and delete when done
with tempfile.NamedTemporaryFile() as temp:
    temp_name = temp.name
    temp.write(b'This file will be deleted automatically\n')
    print(f"Working with temporary file: {temp_name}")
# File is automatically deleted here

# Create a temporary directory
temp_dir = tempfile.mkdtemp()
try:
    print(f"Temporary directory created: {temp_dir}")
    # Use the temporary directory
    with open(os.path.join(temp_dir, 'test.txt'), 'w') as f:
        f.write('Testing temporary directory')
finally:
    # Clean up
    shutil.rmtree(temp_dir)
```

Handling File Locks for Concurrent Access

For multi-process/thread applications:

python

```python
import os
import time
import fcntl  # Unix-based systems only

def acquire_lock(file_obj):
    """
    Acquire an exclusive lock on a file.

    Args:
        file_obj: An open file object

    Returns:
        bool: True if lock acquired, False otherwise
    """
    try:
        fcntl.flock(file_obj, fcntl.LOCK_EX | fcntl.LOCK_NB)
        return True
    except IOError:
        return False

def release_lock(file_obj):
    """
    Release a lock on a file.

    Args:
        file_obj: An open file object with a lock
    """
    fcntl.flock(file_obj, fcntl.LOCK_UN)

# Usage example
def update_counter(counter_file, increment=1):
    """Update a counter in a file with locking."""
    with open(counter_file, 'r+') as file:
        # Try to acquire lock
        if not acquire_lock(file):
            print("File is locked by another process, waiting...")
            while not acquire_lock(file):
                time.sleep(0.1)

        try:
            # Read current value
            current = int(file.read().strip() or '0')
```

```
            # Update the counter
            new_value = current + increment

            # Write the new value
            file.seek(0)
            file.truncate()
            file.write(str(new_value))
            file.flush()

            print(f"Counter updated: {current} -> {new_value}")
            return new_value
        finally:
            # Always release the lock
            release_lock(file)

# Platform-independent locking alternative using a lock file
def with_lock_file(file_path, operation_func):
    """
    Execute a function with a lock file for mutual exclusion.
    Works on both Windows and Unix.

    Args:
        file_path: Path to the file being protected
        operation_func: Function to execute with lock
    """
    lock_path = file_path + '.lock'

    # Try to create the lock file
    try:
        lock_fd = os.open(lock_path, os.O_CREAT | os.O_EXCL | os.O_WRONLY)
    except FileExistsError:
        print("Operation in progress, waiting for lock...")
        # Wait for lock to be released
        while os.path.exists(lock_path):
            time.sleep(0.1)
        # Recursive call now that lock is gone
        return with_lock_file(file_path, operation_func)

    try:
        # Write PID to lock file
        with os.fdopen(lock_fd, 'w') as lock_file:
            lock_file.write(str(os.getpid()))

        # Execute the operation
```

```
        return operation_func(file_path)
    finally:
        # Always remove the lock file
        try:
            os.unlink(lock_path)
        except Exception as e:
            print(f"Warning: Could not remove lock file: {e}")
```

This completes our comprehensive guide to file handling in Python. From basic operations to advanced techniques, you now have a solid foundation for working with files in your Python applications. Remember to follow best practices, handle errors gracefully, and choose the appropriate methods for your specific needs.

Further Reading and References

1. Python Official Documentation:
   - Built-in Functions: open()
   - File Objects
   - The pathlib Module
2. Standard Library Modules:
   - os.path — Common pathname manipulations
   - csv — CSV File Reading and Writing
   - json — JSON encoder and decoder
   - pickle — Python object serialization
   - gzip — Support for gzip files
   - zipfile — Work with ZIP archives
   - tempfile — Generate temporary files and directories