File Operations with Python

Scott Tremaine
Software Developer and Educator

Breakpoint Coding Tutorials

Contents

Working with Files and Exceptions

Introduction to File Handling

File handling is a required skill for any programmer, especially when working with real-world applications. Understanding how to read from and write to files enables you to manage data efficiently, perform persistent storage, and facilitate data exchange between different systems. Files serve as a medium to store data permanently on a storage device, allowing the data to be retained even after the program has finished execution.

Files allow you to share data between different programs. For instance, a program that generates reports might read data from a file created by another program that collects user inputs. Additionally, understanding file handling is fundamental when dealing with configuration files, logging, and serialization. Configuration files store settings and preferences, which your program can read upon startup to configure itself accordingly. Logging is essential for tracking the program's behavior, debugging, and auditing, which is done by writing logs to files. Serialization, the process of converting an object into a format that can be easily stored and retrieved, often uses file handling to persist objects' states.

The importance of file handling extends to data analysis and scientific computing as well. Analysts and scientists frequently deal with large datasets stored in files, which they need to process and analyze using their programs. Without file handling, managing and utilizing such data would be nearly impossible.

File Types: Text and Binary

When dealing with files in Python, it is essential to understand the two main types: text files and binary files. Each type has its characteristics and uses, influencing how you read from and write to these files.

Text Files

Text files store data in a human-readable format, typically consisting of characters encoded using a character encoding standard such as ASCII or UTF-8. These files are plain and can be opened and edited using any text editor. Common examples include files with extensions like .txt, .csv, .html, and .json.

Text files are used for a wide range of purposes, such as configuration files, logs, source code files, and more. When you read a text file, you retrieve data as strings, and writing to a text file involves converting your data into a string format.

Here's an example of how you might read from and write to a text file in Python:

```
# Writing to a text file
with open('example.txt', 'w') as file:
    file.write("Hello, World!\n")
    file.write("This is a text file.")

# Reading from a text file
with open('example.txt', 'r') as file:
    content = file.read()
```

```
    print(content)
```

In this example, the `with open()` statement is used to open the file, ensuring that it is properly closed after the operations are completed. The `write()` method writes strings to the file, while the `read()` method reads the entire content of the file as a single string.

Binary Files

Binary files store data in a format that is not intended to be human-readable. Instead of characters, binary files contain bytes, which can represent any type of data, including text, images, audio, and video. Examples of binary files include files with extensions like .bin, .jpg, .mp3, and .dat.

Reading from and writing to binary files involves handling data as bytes rather than strings. This is crucial when working with non-text data to ensure that it is stored and retrieved accurately without any alteration.

Here's an example of how to work with binary files in Python:

```
# Writing to a binary file
data = b'\x48\x65\x6c\x6c\x6f\x2c\x20\x57\x6f\x72\x6c\x64\x21'
with open('example.bin', 'wb') as file:
    file.write(data)

# Reading from a binary file
with open('example.bin', 'rb') as file:
    binary_content = file.read()
    print(binary_content)
```

In this example, the `wb` mode is used for writing binary data, and the `rb` mode is used for reading binary data. The data is written and read as bytes, ensuring that the original binary content is preserved.

Choosing Between Text and Binary Files

The choice between text and binary files depends on the nature of the data and the intended use. Text files are ideal for storing human-readable information, such as configuration settings, logs, and documentation. They are easy to edit and review using standard text editors.

Binary files, on the other hand, are suitable for storing non-text data, such as multimedia content, serialized objects, and data in proprietary formats. They are more efficient in terms of storage and performance for these types of data, as they do not require encoding and decoding like text files.

Understanding the differences between text and binary files and knowing when to use each type is crucial for effective file handling. It ensures that your data is stored and processed in the most appropriate format, leading to better performance and reliability in your applications.

Reading and Writing Files

Interacting with files is a fundamental skill that every developer should learn.

Opening and Closing Files

Opening and closing files in Python is straightforward, thanks to the built-in `open()` function. When you open a file, Python creates a file object, which provides methods and attributes to interact with the file's content.

To open a file, you use the `open()` function, which takes two main arguments: the filename and the mode. The mode determines how the file will be used, such as for reading, writing, or appending.

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

In the example above, `'example.txt'` is the name of the file, and `'r'` stands for "read mode." Here are the common file modes:

- `'r'`: Read (default mode) - Opens the file for reading.
- `'w'`: Write - Opens the file for writing (creates a new file or truncates the existing file).
- `'a'`: Append - Opens the file for appending (creates a new file if it doesn't exist).
- `'b'`: Binary mode - Used with other modes for binary files.
- `'+'`: Read and write - Used with other modes for both reading and writing.

To ensure that a file is properly closed after its operations, you should use the `close()` method. However, a more convenient and safer way is to use a `with` statement, which automatically closes the file when the block is exited.

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

The `with` statement is highly recommended because it ensures that the file is properly closed even if an exception occurs.

Reading from Files

Reading data from a file can be done in several ways, depending on your needs. The most common methods are `read()`, `readline()`, and `readlines()`.

- `read(size=-1)`: Reads the entire file or up to the specified number of bytes if the `size` argument is provided.
- `readline(size=-1)`: Reads a single line from the file. If the `size` argument is provided, it reads up to that number of bytes from the line.
- `readlines(hint=-1)`: Reads all lines from the file and returns them as a list. If the `hint` argument is provided, it reads that many lines.

Here is an example of how to use these methods:

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

This code reads the entire content of `example.txt` and prints it to the console. If the file is large, you might want to read it line by line:

```
with open('example.txt', 'r') as file:
    for line in file:
        print(line, end='')
```

This approach is memory efficient as it processes one line at a time.

Writing to Files

Writing data to a file is just as important as reading from it. Python provides several methods for writing to files: `write()` and `writelines()`.

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

When writing to a file, you typically use either the `'w'` mode to start with a fresh file or the `'a'` mode to append to an existing file.

Here's an example of writing to a file using the `write()` method:

```
with open('example.txt', 'w') as file:
    file.write("Hello, World!\n")
    file.write("This is a test file.")
```

This code creates (or overwrites) `example.txt` and writes two lines of text to it. If you want to append to the file instead of overwriting it, you can use the `'a'` mode:

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

For writing multiple lines at once, use the `writelines()` method:

```
lines = ["First line\n", "Second line\n", "Third line\n"]
with open('example.txt', 'w') as file:
    file.writelines(lines)
```

This code writes a list of lines to `example.txt` in one go.

Working with File Paths

Managing file paths is crucial, especially when dealing with files located in different directories. Python's `os` and `pathlib` modules offer tools to handle file paths effectively.

Using the `os` module

The `os` module provides functions to interact with the operating system. You can use `os.path` to handle file paths.

```
import os

file_path = os.path.join('folder', 'subfolder', 'example.txt')
print(file_path)  # Output: folder/subfolder/example.txt
```

Using the `pathlib` module

The `pathlib` module offers an object-oriented approach to handling file paths, making the code more readable and expressive.

```
from pathlib import Path

file_path = Path('folder') / 'subfolder' / 'example.txt'
print(file_path)  # Output: folder/subfolder/example.txt
```

`pathlib` is preferred for its simplicity and modern interface. It also provides methods for common file operations, such as checking if a file exists, reading and writing files, and more.

```
file_path = Path('example.txt')

# Check if the file exists
if file_path.exists():
    print("File exists.")
else:
    print("File does not exist.")

# Read the file
if file_path.is_file():
    with file_path.open('r') as file:
        content = file.read()
        print(content)
```

Working with Different File Formats

Two of the most common formats you'll encounter are CSV and JSON. Understanding how to work with these formats will enhance your ability to manage data effectively and integrate with other systems and applications.

Handling CSV Files

CSV, or Comma-Separated Values, is a simple file format used to store tabular data, such as a spreadsheet or database. A CSV file consists of lines of text where each line represents a row in the table, and each value in the row is separated by a comma. This format is widely supported and easy to process, making it a popular choice for data exchange.

Understanding the Structure of a CSV File

A CSV file typically looks like this:

```
Name,Age,Occupation
John Doe,30,Software Engineer
Jane Smith,25,Data Scientist
```

Each line represents a record, and the fields within a record are separated by commas. The first line often contains the header, which labels each column.

Reading CSV Files

Python's built-in csv module provides functionality to read and write CSV files efficiently. To read a CSV file, you can use the csv.reader object. Here's how you can do it:

```
import csv

with open('data.csv', mode='r') as file:
    csv_reader = csv.reader(file)
    header = next(csv_reader)  # Read the header
    for row in csv_reader:
        print(row)
```

In this example, we open the CSV file in read mode and create a csv_reader object. The next function reads the header row, and then we iterate over the remaining rows to process the data.

Writing CSV Files

To write data to a CSV file, you use the csv.writer object. Here's an example:

```
import csv

data = [
    ['Name', 'Age', 'Occupation'],
    ['John Doe', 30, 'Software Engineer'],
    ['Jane Smith', 25, 'Data Scientist']
]

with open('data.csv', mode='w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerows(data)
```

In this code snippet, we define the data as a list of lists and use the csv_writer to write the data to a file. The writerows method writes all rows to the file.

Using DictReader and DictWriter

The csv module also provides DictReader and DictWriter for working with CSV files as dictionaries, which can be more intuitive when dealing with named columns.

```
import csv

with open('data.csv', mode='r') as file:
    csv_reader = csv.DictReader(file)
    for row in csv_reader:
        print(row['Name'], row['Age'], row['Occupation'])
```

Here, DictReader reads each row into an OrderedDict, allowing you to access values by their column names.

For writing, you can use DictWriter:

```
import csv

data = [
    {'Name': 'John Doe', 'Age': 30, 'Occupation': 'Software Engineer'},
    {'Name': 'Jane Smith', 'Age': 25, 'Occupation': 'Data Scientist'}
]

with open('data.csv', mode='w', newline='') as file:
    fieldnames = ['Name', 'Age', 'Occupation']
    csv_writer = csv.DictWriter(file, fieldnames=fieldnames)
    csv_writer.writeheader()
    csv_writer.writerows(data)
```

In this example, DictWriter writes dictionaries to the CSV file, ensuring that the fieldnames are used as headers.

Best Practices for Handling CSV Files

- Always specify the newline parameter when opening a file in write mode to avoid extra blank lines on Windows.
- Use DictReader and DictWriter for more readable and maintainable code when working with CSV files that have headers.
- Handle exceptions using try-except blocks to manage errors such as file not found or permission issues.

Handling JSON Files

JSON, or JavaScript Object Notation, is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. JSON is widely used for data exchange in web applications and APIs due to its simplicity and flexibility.

Understanding the Structure of a JSON File

A JSON file represents data as nested key-value pairs, arrays, and other JSON objects. Here's an example of a JSON structure:

```
{
    "employees": [
        {
            "name": "John Doe",
            "age": 30,
            "occupation": "Software Engineer"
        },
        {
            "name": "Jane Smith",
            "age": 25,
            "occupation": "Data Scientist"
```

```
        }
    ]
}
```

In this example, employees is an array of objects, each representing an employee with their respective attributes.

Reading JSON Files

Python provides the json module to read and write JSON data. To read a JSON file, you can use the json.load function:

```
import json

with open('data.json', mode='r') as file:
    data = json.load(file)
    for employee in data['employees']:
        print(employee['name'], employee['age'], employee['occupation']
    ])
```

This code opens a JSON file in read mode and loads the data into a Python dictionary. We then iterate over the employees array to access each employee's details.

Writing JSON Files

To write data to a JSON file, you use the json.dump function. Here's an example:

```
import json

data = {
    "employees": [
        {"name": "John Doe", "age": 30, "occupation": "Software
    Engineer"},
        {"name": "Jane Smith", "age": 25, "occupation": "Data Scientist
    "}
    ]
}

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

In this code snippet, we define the data as a dictionary and use json.dump to write it to a file. The indent parameter makes the JSON output more readable.

Parsing JSON Strings

Sometimes, you may need to parse JSON data from a string instead of a file. You can use json.loads for this purpose:

```
import json

json_string = '''
{
    "employees": [
        {"name": "John Doe", "age": 30, "occupation": "Software
    Engineer"},
```

```
        {"name": "Jane Smith", "age": 25, "occupation": "Data Scientist
    "}
    ]
}
'''

data = json.loads(json_string)
for employee in data['employees']:
    print(employee['name'], employee['age'], employee['occupation'])
```

This example demonstrates how to parse a JSON string into a Python dictionary and iterate over the data.

Best Practices for Handling JSON Files

- Always use the indent parameter in json.dump for better readability of JSON files.
- Handle exceptions using try-except blocks to manage errors such as file not found, JSON decoding errors, or permission issues.
- Use json.loads and json.dumps for parsing and generating JSON data from strings.

Exception Handling

Exception handling is a critical aspect of writing robust and reliable software. By anticipating and managing potential errors, you can ensure that your programs behave gracefully under unexpected conditions. Python provides a rich set of tools for handling exceptions, allowing you to capture, respond to, and even create custom error conditions in your code.

Understanding Exceptions

In Python, an exception is an event that disrupts the normal flow of a program. When an error occurs within a function or a block of code, Python raises an exception, signaling that something has gone wrong. Exceptions can be caused by various issues, such as invalid input, division by zero, or attempts to access nonexistent files.

To illustrate, consider the following example where we attempt to divide a number by zero:

```
def divide(a, b):
    return a / b

result = divide(10, 0)
```

Running this code results in a ZeroDivisionError because dividing by zero is undefined in mathematics and disallowed in Python.

Common Types of Exceptions

Python has numerous built-in exceptions, each serving different purposes. Some of the most common ones include:

- **ValueError:** Raised when a function receives an argument of the right type but an inappropriate value.
- **TypeError:** Raised when an operation or function is applied to an object of inappropriate type.
- **IndexError:** Raised when trying to access an element from a list or tuple at an invalid index.
- **KeyError:** Raised when attempting to access a dictionary with a key that doesn't exist.
- **FileNotFoundError:** Raised when trying to open a file that cannot be found.

Understanding these exceptions helps you anticipate and handle errors effectively in your code.

The `try`, `except`, `else`, and `finally` Blocks

Python's exception handling mechanism revolves around the `try` and `except` blocks. The `try` block contains the code that might raise an exception, while the `except` block handles the exception if it occurs.

Consider this example where we handle a potential division by zero error:

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
        return None
    else:
        return result
    finally:
        print("Execution completed.")

result = divide(10, 0)
```

Explanation of the Blocks

- `try`: The code within this block is executed first. If no exceptions occur, the `except` block is skipped.
- `except`: If an exception occurs in the `try` block, Python jumps to this block. You can specify the type of exception to catch. In this example, we catch a ZeroDivisionError.
- `else`: This block is optional and executes if the code in the `try` block runs without any exceptions. It's useful for code that should only run if no exceptions occur.
- `finally`: This block is also optional and executes regardless of whether an exception occurs or not. It's typically used for cleanup actions, such as closing files or releasing resources.

Catching Specific Exceptions

Catching specific exceptions allows you to handle different types of errors in different ways, improving the robustness and user-friendliness of your code. You can specify multiple `except` blocks to handle various exceptions separately.
Here's an example that demonstrates catching multiple specific exceptions:

```
def handle_errors():
    try:
        value = int(input("Enter a number: "))
        result = 10 / value
        print("Result:", result)
    except ValueError:
        print("Error: Invalid input. Please enter a valid integer.")
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
    except Exception as e:
        print(f"Unexpected error: {e}")
    finally:
        print("Execution completed.")

handle_errors()
```

In this example:

- We catch `ValueError` if the input is not an integer.
- We catch `ZeroDivisionError` if the input is zero.
- We catch any other unexpected exceptions using a generic `Exception` block.

This approach ensures that different error conditions are managed appropriately, providing clear feedback to the user.

Raising Exceptions

Sometimes, you might want to raise exceptions deliberately to signal that an error condition has occurred. This is especially useful when validating input or enforcing certain constraints in your code.

To raise an exception, use the `raise` statement followed by the exception type. You can also include an error message for more context.

```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    elif age > 120:
        raise ValueError("Age seems unrealistic.")
    else:
        print("Age is valid.")

try:
    validate_age(-5)
except ValueError as e:
    print(f"Validation error: {e}")
```

In this example, the `validate_age` function raises a `ValueError` if the age is negative or unrealistically high. The `try` block catches the exception and prints an error message.

Raising exceptions allows you to enforce rules and constraints in your code, ensuring that only valid data is processed further.

Custom Exceptions

While Python provides many built-in exceptions, you might encounter situations where you need to create your own custom exceptions. Custom exceptions allow you to define error conditions specific to your application's logic, making your code more readable and maintainable.

To create a custom exception, define a new class that inherits from Python's built-in `Exception` class or any of its subclasses.

Here's an example of defining and using a custom exception:

```
class InvalidAgeError(Exception):
    def __init__(self, age, message="Invalid age provided."):
        self.age = age
        self.message = message
        super().__init__(self.message)

def validate_age(age):
    if age < 0 or age > 120:
        raise InvalidAgeError(age)

try:
    validate_age(150)
except InvalidAgeError as e:
    print(f"Error: {e}")
```

In this example:

- We define a custom exception `InvalidAgeError` that inherits from `Exception`.
- The custom exception includes an initializer to accept the age and an optional message.
- The `validate_age` function raises `InvalidAgeError` if the age is invalid.
- The `try` block catches the custom exception and prints the error message.

Custom exceptions provide a clear and consistent way to handle specific error conditions in your application, enhancing code readability and maintainability.

Practical Applications of File Handling and Exceptions

Reading and Writing Config Files

Configuration files are a common way to store settings and parameters for an application. They allow you to separate the configuration from the code, making it easier to manage

and modify settings without changing the actual program.

Understanding Config Files

Configuration files can be in various formats, such as INI, JSON, or YAML. The choice of format depends on the complexity and requirements of your application. For simplicity, we will focus on INI files, which are widely used and easy to work with.

An INI file typically looks like this:

```
[settings]
database = my_database
user = admin
password = secret

[paths]
logfile = /var/log/myapp.log
```

Here, the file is divided into sections ([settings] and [paths]), each containing key-value pairs.

Reading Config Files

Python's configparser module makes it easy to read and write INI files. Here's how you can read a config file:

```
import configparser

config = configparser.ConfigParser()
config.read('config.ini')

database = config['settings']['database']
user = config['settings']['user']
password = config['settings']['password']
logfile = config['paths']['logfile']

print(f"Database: {database}, User: {user}, Logfile: {logfile}")
```

In this example, we create a ConfigParser object and read the config file. We then access the settings and paths using the section and key names.

Writing Config Files

To write data to a config file, you can use the ConfigParser object as well:

```
import configparser

config = configparser.ConfigParser()

config['settings'] = {
    'database': 'my_database',
    'user': 'admin',
    'password': 'secret'
}

config['paths'] = {
    'logfile': '/var/log/myapp.log'
```

```
}

with open('config.ini', 'w') as configfile:
    config.write(configfile)
```

Here, we define the sections and keys in a dictionary format and write them to a config file using the write method.

Best Practices for Config Files

- **Keep sensitive information secure:** Avoid storing plain text passwords in config files. Use environment variables or secure vaults.
- **Validate configuration data:** Ensure the configuration values are valid and within expected ranges to prevent runtime errors.
- **Provide default values:** Use default values for settings to ensure the application can run with minimal configuration.

Logging with Files

Logging is an essential part of any application. It helps track the flow of the program, debug issues, and monitor application performance. Python's logging module provides a flexible framework for emitting log messages from Python programs.

Setting Up Basic Logging

Here's how you can set up basic logging to a file:

```
import logging

logging.basicConfig(filename='app.log', level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')

logging.info('This is an informational message.')
logging.warning('This is a warning message.')
logging.error('This is an error message.')
```

In this example, we configure the logging to write messages to app.log. The log level is set to INFO, which means all messages at this level and above will be logged. The format specifies how the log messages will be displayed, including the timestamp, log level, and message.

Advanced Logging Configuration

For more complex logging requirements, you can use a configuration file or dictionary to set up the logging. This allows for greater flexibility and maintainability.

```
import logging.config

log_config = {
    'version': 1,
    'formatters': {
        'default': {
```

```
            'format': '%(asctime)s - %(name)s - %(levelname)s - %(
    message)s',
        },
    },
    'handlers': {
        'file': {
            'class': 'logging.FileHandler',
            'filename': 'app.log',
            'formatter': 'default',
        },
        'console': {
            'class': 'logging.StreamHandler',
            'formatter': 'default',
        },
    },
    'root': {
        'level': 'INFO',
        'handlers': ['file', 'console']
    }
}

logging.config.dictConfig(log_config)

logging.info('Logging is configured via dictionary.')
```

This configuration logs messages to both a file and the console, making it easier to monitor the application during development and production.

Best Practices for Logging

- **Use appropriate log levels:** Use DEBUG for development details, INFO for general information, WARNING for potential issues, ERROR for errors, and CRITICAL for severe errors.
- **Avoid logging sensitive information:** Ensure that sensitive data, such as passwords and personal information, are not logged.
- **Rotate log files:** Use log rotation to prevent log files from growing indefinitely and consuming disk space. Python's logging.handlers module provides RotatingFileHandler for this purpose.

Error Handling in Real-World Applications

Handling errors gracefully is crucial for building robust applications. Effective error handling ensures that your program can recover from unexpected issues and provide meaningful feedback to the user.

Basic Exception Handling

Let's start with a simple example of handling a file operation error:

```
try:
    with open('non_existent_file.txt', 'r') as file:
        data = file.read()
except FileNotFoundError:
```

```
    print("File not found. Please check the file path.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

In this example, we use a try block to attempt to open and read a file. If the file does not exist, a FileNotFoundError is raised, and we handle it by printing a user-friendly message. The generic Exception catch-all handles any other unexpected errors.

Custom Exception Classes

For more complex applications, defining custom exception classes can help manage specific error conditions more effectively:

```
class ConfigurationError(Exception):
    pass

def load_config(file_path):
    try:
        with open(file_path, 'r') as file:
            # Simulating a configuration load failure
            raise ConfigurationError("Failed to load configuration.")
    except ConfigurationError as e:
        print(f"Configuration error: {e}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

load_config('config.ini')
```

Here, we define a custom ConfigurationError and raise it if the configuration file fails to load. This approach makes it easier to identify and handle specific error conditions.

Logging Exceptions

Combining logging with exception handling provides a powerful way to track and diagnose issues in your application. Here's an example:

```
import logging

logging.basicConfig(filename='app.log', level=logging.ERROR,
                    format='%(asctime)s - %(levelname)s - %(message)s')

def divide(a, b):
    try:
        result = a / b
        return result
    except ZeroDivisionError as e:
        logging.error(f"Error occurred: {e}")
        print("Cannot divide by zero.")
    except Exception as e:
        logging.error(f"Unexpected error: {e}")
        print("An unexpected error occurred.")

divide(10, 0)
```

In this example, we log any exceptions that occur during the division operation. This way, we can keep track of errors and investigate them later using the log file.

Best Practices for Error Handling

- **Be specific with exceptions:** Catch specific exceptions whenever possible to handle different error conditions appropriately.
- **Provide meaningful messages:** Ensure that error messages are clear and provide enough information to understand the issue.
- **Clean up resources:** Use finally blocks to clean up resources such as file handles and network connections, ensuring they are closed properly even if an error occurs.
- **Fail gracefully:** Design your application to continue functioning or shut down gracefully in the event of an error, providing a good user experience.