Day 30 Learning Summary

Errors, Exception Handling & JSON Data Processing

100 Days of Code Challenge

Section 30 - September 12, 2025

Contents

1 Overview

Day 30 focused on two critical aspects of Python programming:

- **Exception Handling**: Learning to gracefully handle errors and prevent program crashes
- **JSON Data Processing**: Working with JSON strings and files for data exchange

2 Exception Handling in Python

2.1 The Need for Exception Handling

Exception handling is essential for creating robust applications that can:

- Handle runtime errors gracefully
- Prevent unexpected program termination
- Provide meaningful feedback to users
- Maintain program flow even when errors occur

2.2 Try-Except Structure

The fundamental structure for handling exceptions in Python follows this pattern:

```
try:
    # Code that might cause an exception
    risky_operation()
except SpecificError:
    # Handle specific error type
    handle_error()
except AnotherError:
    # Handle another error type
    handle_another_error()
else:
    # Execute if no exceptions occurred
    success_operations()
finally:
    # Always execute (cleanup operations)
    cleanup()
```

2.3 Key Components

2.3.1 Try Block

Contains code that might raise an exception. All potentially risky operations should be placed here.

2.3.2 Except Block

Handles specific exceptions. **Important**: Never use bare `except` clauses as they can mask other errors.

Bad Practice:

```
try:
    file = open('missing_file.txt')
except:  # This catches ALL exceptions
    print("Something went wrong")
```

Good Practice:

```
try:
    file = open('missing_file.txt')
except FileNotFoundError as e:
    print(f"File not found: {e}")
except PermissionError as e:
    print(f"Permission denied: {e}")
```

2.3.3 Else Block

Executes only when no exceptions occur in the try block. Used for operations that should only run on success.

2.3.4 Finally Block

Always executes regardless of whether an exception occurred. Primarily used for cleanup operations like:

- Closing files
- Releasing resources
- Database connections cleanup

2.4 Common Exception Types

- `FileNotFoundError`: Raised when trying to open a non-existent file
- `KeyError`: Raised when accessing a non-existent dictionary key
- `ValueError`: Raised when a function receives an argument of correct type but inappropriate value
- `TypeError`: Raised when an operation is performed on inappropriate type

2.5 Raising Custom Exceptions

The `raise` keyword allows you to trigger exceptions manually:

```
height = float(input("Height in meters: "))
weight = int(input('Weight: '))

if height > 3.0:
    raise ValueError('Please enter height less than 3m!')

bmi = weight / height ** 2
```

This is useful for:

- Input validation
- Business logic enforcement
- Creating meaningful error messages

3 JSON Data Processing

3.1 Introduction to JSON

JSON (JavaScript Object Notation) is a lightweight data interchange format that is:

- Human-readable
- Language-independent
- Widely used for APIs and configuration files
- Similar to Python dictionaries in structure

3.2 Python's JSON Module

Python provides the built-in `json` module with key methods:

- `json.loads()`: Parse JSON string to Python object
- `json.dumps()`: Convert Python object to JSON string
- `json.load()`: Read JSON from file
- `json.dump()`: Write JSON to file

3.3 Working with JSON Strings

3.3.1 Parsing JSON Strings

```
import json

json_string = '''
{
    "students": [
        {
            "id": 1,
            "name": "Tim",
            "age": 21,
            "full-time": true
        },
        {
            "id": 2,
            "name": "Joe",
            "age": 33,
            "full-time": false
        }
    ]
}
'''

# Convert JSON string to Python dictionary
data = json.loads(json_string)
print(type(data))  # <class 'dict'>
```

3.3.2 Converting to JSON

```
# Add new data to the dictionary
data['State'] = True

# Convert Python object back to JSON string
new_json = json.dumps(data, indent=4)
print(new_json)
```

3.4 Working with JSON Files

3.4.1 Reading JSON Files

```
with open('./data.json', 'r') as f:
    data = json.load(f)  # Returns a dictionary
print(data)
```

3.4.2 Writing to JSON Files

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

3.4.3 Appending Data to JSON Files

To append new data to an existing JSON file:

```
# 1. Read existing data
with open('data.json', 'r') as f:
    data = json.load(f)

# 2. Modify the data
new_student = {
    "id": 3,
    "name": "Anna",
    "age": 25,
    "full-time": True
}
data['students'].append(new_student)

# 3. Write back to file
with open('data.json', 'w') as f:
    json.dump(data, f, indent=4)
```

4 Best Practices Learned

4.1 Exception Handling Best Practices

- Always specify exception types in `except` blocks
- Use multiple `except` blocks for different error types
- Utilize `finally` blocks for cleanup operations
- Raise custom exceptions with meaningful messages
- Log errors appropriately for debugging

4.2 JSON Processing Best Practices

- Use `indent` parameter for readable JSON output
- Always use context managers (`with` statement) for file operations
- Validate JSON structure before processing
- Handle potential JSON parsing errors with try-except blocks

5 Practical Applications

5.1 File Operations with Error Handling

The combination of exception handling and file operations ensures robust applications:

```
try:
    with open('data.json', 'r') as f:
        data = json.load(f)
except FileNotFoundError:
    print("File not found, creating new one...")
    data = {"students": []}
    with open('data.json', 'w') as f:
        json.dump(data, f, indent=4)
except json.JSONDecodeError:
    print("Invalid JSON format in file")
else:
    print("Data loaded successfully")
finally:
    print("File operation completed")
```

6 Key Takeaways

1. **Defensive Programming**: Always anticipate potential errors and handle them gracefully

2. **Resource Management**: Use `finally` blocks and context managers for proper cleanup
3. **Specific Error Handling**: Target specific exception types rather than using bare except clauses
4. **Data Validation**: Validate input data and raise meaningful exceptions for invalid inputs
5. **JSON Integration**: JSON is essential for modern applications and data exchange
6. **File Safety**: Always handle file operations with proper exception handling

7 Conclusion

Day 30 provided essential skills for building robust Python applications. Exception handling ensures programs can gracefully recover from errors, while JSON processing enables effective data exchange and storage. These concepts are fundamental for professional software development and form the backbone of reliable, user-friendly applications.

The practical exercises demonstrated real-world scenarios where these concepts are applied, from file operations to data validation, preparing for more advanced topics in web development, API integration, and data processing.