Exception Handling, Real-World Scenarios & Updating JSON

AiCE Software Engineering Foundations — Python Programming (Advanced)

This closing lesson on JSON covers handling the ways JSON operations can fail, two real-world scenarios — API responses and configuration files — and the load-modify-save pattern used to update stored JSON data. It closes this unit and prepares you directly for the Student Management System capstone ahead.

1. Exception Handling with JSON

JSON operations can fail in several ways — the file might not exist, the JSON might be malformed, or a required key might be missing. Wrapping JSON operations in exception handling is essential for production-quality code:

```
import json
import os

def safe_load_json(filename):
    """Load JSON from a file with full exception handling.
    Returns (data, error_message).
    """
    try:
        if not os.path.exists(filename):
            return [], None    # Empty list is valid for a new
database
        with open(filename, "r") as file:
            data = json.load(file)
        return data, None
    except json.JSONDecodeError as e:
        return None, f"Corrupt JSON file: {e}"
    except PermissionError:
        return None, f"Permission denied reading {filename}"
    except Exception as e:
        return None, f"Unexpected error: {e}"
```

```
def safe_save_json(filename, data):
    """Save data to a JSON file with full exception handling.
    Returns (True, None) on success or (False, error_message)
on failure.
    """
    try:
        with open(filename, "w") as file:
            json.dump(data, file, indent=4)
        return True, None
    except PermissionError:
        return False, f"Permission denied writing to
{filename}"
    except TypeError as e:
        return False, f"Data contains non-serialisable type:
{e}"
    except Exception as e:
        return False, f"Unexpected error: {e}"

# Test the safe functions
data, error = safe_load_json("aice_students.json")
if error:
    print(f"Load error: {error}")
else:
    print(f"Loaded {len(data)} records successfully.")

success, error = safe_save_json("backup.json", data)
if error:
    print(f"Save error: {error}")
else:
    print("Backup saved successfully.")

# Test with corrupt JSON
with open("corrupt.json", "w") as f:
    f.write("{this is not valid json")

data2, error2 = safe_load_json("corrupt.json")
if error2:
    print(f"\nExpected error: {error2}")
```

json.JSONDecodeError is the specific exception raised when JSON parsing fails — for example, if the file was partially written or manually corrupted. Catching it specifically, rather than a bare Exception, gives you a precise, informative error message, following the result-or-error return pattern from the Exception Handling unit.

2. JSON in Real-World Scenarios

Reading a JSON API Response

When you make HTTP requests to web APIs, the responses come as JSON strings. json.loads() converts them directly to Python objects:

```
import json

# Simulated API response (as a string)
api_response = """
{
    "status": "success",
    "data": {
        "platform": "AiCE",
        "total_students": 1200,
        "countries": ["Nigeria", "Ghana", "Kenya", "Senegal"],
        "latest_cohort": {
            "id": "COHORT-2026-A",
            "enrolled": 85,
            "pass_rate": 0.78
        }
    }
}
"""

# Parse the response
response = json.loads(api_response)

# Extract nested data
print(response["status"])
print(response["data"]["platform"])
print(response["data"]["total_students"])
print(response["data"]["countries"])
```

```
cohort = response["data"]["latest_cohort"]
print(f"Cohort   : {cohort['id']}")
print(f"Enrolled : {cohort['enrolled']}")
print(f"Pass rate: {cohort['pass_rate']:.0%}")
```

Configuration Files in JSON

JSON is commonly used for application configuration — storing settings that can be changed without modifying the code:

```
import json

# Write a configuration file
config = {
    "platform"     : "AiCE",
    "version"      : "2.0.0",
    "database_file": "students.json",
    "passing_mark" : 50,
    "distinction"  : 70,
    "max_students" : 500,
    "features"     : {
        "colour_output": True,
        "auto_backup"  : True,
        "backup_count" : 5
    }
}

with open("config.json", "w") as f:
    json.dump(config, f, indent=4)

# Load and use configuration
with open("config.json", "r") as f:
    cfg = json.load(f)

print(f"Platform : {cfg['platform']} v{cfg['version']}")
print(f"Database : {cfg['database_file']}")
print(f"Pass mark: {cfg['passing_mark']}")
print(f"Colour   : {cfg['features']['colour_output']}")
```

Reading settings from a config.json file rather than hardcoding them in your program makes your application much more flexible. Change the passing mark? Update config.json. Add a new feature flag? Add it to config.json. No code changes needed.

> **The Handshake Between Systems**
> *"JSON is the handshake between systems. Your Python program, a web API, a mobile app, a JavaScript frontend — they speak different languages, but they all understand JSON. Master it and your programs can talk to the world." — Adesina Olagunju, Founder, AiCE*

3. Updating JSON Data

Updating JSON data follows the same pattern as any file operation — load the data, modify it in memory, save it back:

```
import json
import os

DATABASE = "aice_students.json"

def load_data():
    if not os.path.exists(DATABASE):
        return []
    with open(DATABASE, "r") as f:
        return json.load(f)

def save_data(data):
    with open(DATABASE, "w") as f:
        json.dump(data, f, indent=4)

# Load existing data
students = load_data()
print(f"Before update: {len(students)} students")

# Add a new student
new_student = {
    "name": "Zainab Musa", "age": 20, "country": "Nigeria",
    "scores": {"Math": 78, "English": 82, "Science": 75}
}
```

```
students.append(new_student)

# Update an existing student (first student)
if students:
    students[0]["scores"]["Math"] = 95    # Update a score

# Remove a student by name
students = [s for s in students if s["name"] != "Kofi Mensah"]

# Save the updated data
save_data(students)
print(f"After update : {len(students)} students")

# Verify
verified = load_data()
for s in verified:
    avg = sum(s["scores"].values()) / len(s["scores"])
    print(f"  {s['name']:<22} | Avg: {avg:.1f}")
```

The pattern is always the same: load the full dataset into memory, perform your operations (add, update, or remove items), then save the entire dataset back. This works well for small to medium datasets — the kind in a student management system. For very large datasets, a proper database system would be more appropriate.

Key Takeaways

json.JSONDecodeError is the specific exception for malformed JSON, and wrapping load/save operations in the result-or-error pattern keeps a program resilient to missing files, corrupt data, and permission issues. JSON's real-world reach spans API responses, parsed directly with json.loads(), and configuration files, which make an application's behavior adjustable without touching code. Updating stored JSON always follows load → modify in memory → save — the exact pattern behind adding, updating, and removing records in any JSON-backed application. With JSON fully covered — alongside functions, scope, modules, file handling, exceptions, comprehensions, tuples and sets, OOP, inheritance, and external libraries — you now have every tool needed for the culminating project of this program: the AiCE Student Management System.