Day 3 Session 4:
File I/O, Unit Testing & Debugging - Deep Dive

This notebook is critical for building robust, verifiable, and maintainable Python applications. We will explore how to interact with files to store and retrieve data, write automated tests to ensure code correctness, and effectively debug issues when they arise. These skills are fundamental for any serious engineering or scientific software project.

1. File I/O: Reading and Writing Data

File Input/Output (I/O) is the process of reading data from a file or writing data to a file. It's essential for persisting data, loading configurations, and exchanging information with other systems.

1.1 Text Files: Read and Write

Text files are the simplest form of data storage. Python provides built-in functions to handle them.

Conceptual Approach:

Files are opened using the `open()` function, which returns a file object. You then read from or write to this object. It's crucial to always close the file after use to free up system resources. The `with` statement (covered in Section 2) is the preferred way to handle files, as it ensures automatic closing.

File Opening Modes

Mode	Description	Action if File Exists	Action if File Missing	Read/Write
'r'	Read (default)	Opens ▾	Error (FileNo... ▾	Read ▾
'w'	Write (creates new file or truncates existing)	Truncates to 0 byt... ▾	Creates ▾	Write ▾

'a'	Append (writes to end of file)	Opens ▾	Creates ▾	Write ▾
'x'	Exclusive creation (fails if file exists)	Error (FileExistsEr... ▾	Creates ▾	Write ▾
'b'	Binary mode (for non-text files)	(Used with other ... ▾	(Used with ot... ▾	Read/... ▾
'+'	Open for updating (read and write)	Opens ▾	Creates ▾	Read/... ▾

Code Implementation: Basic Text File Operations

```python
# --- Writing to a text file ---
file_path_txt = "sensor_log.txt"
sensor_readings_to_write = [
    "2025-07-01 10:00:01,TEMP,25.5\n",
    "2025-07-01 10:00:02,PRES,101.2\n",
    "2025-07-01 10:00:03,FLOW,12.3\n"
]

print("--- Writing to Text File ---")
try:
    # 'w' mode truncates the file if it exists, then writes
    with open(file_path_txt, 'w') as f:
        f.write("--- Sensor Data Log ---\n")
        f.writelines(sensor_readings_to_write)
    print(f"Successfully wrote to '{file_path_txt}'.")
except IOError as e:
    print(f"Error writing to file: {e}")

# --- Appending to a text file ---
print("\n--- Appending to Text File ---")
new_entry = "2025-07-01 10:00:04,HUM,60.1\n"
try:
    # 'a' mode appends to the end of the file
    with open(file_path_txt, 'a') as f:
        f.write(new_entry)
    print(f"Successfully appended to '{file_path_txt}'.")
```

```python
    except IOError as e:
        print(f"Error appending to file: {e}")

    # --- Reading from a text file (full content) ---
    print("\n--- Reading from Text File ---")
    try:
        with open(file_path_txt, 'r') as f:
            full_content = f.read()
            print("\nFull Content:")
            print(full_content)
    except FileNotFoundError:
        print(f"Error: File '{file_path_txt}' not found.")
    except IOError as e:
        print(f"Error reading file: {e}")

    # --- Reading line by line ---
    print("\n--- Reading Line by Line ---")
    try:
        with open(file_path_txt, 'r') as f:
            for line_num, line in enumerate(f, start=1):
                print(f"Line {line_num}: {line.strip()}")
    except FileNotFoundError:
        print(f"Error: File '{file_path_txt}' not found.")
```

1.2 CSV (Comma Separated Values) Files

CSV files are a common format for tabular data. Python's built-in `csv` module provides robust functionality for reading and writing CSV data, handling quoting and delimiters correctly.

Conceptual Approach:

Use `csv.reader` to iterate over lines in a CSV file and `csv.writer` to write lists of data to a CSV file.

Code Implementation: CSV File Operations.

```python
import csv

csv_file_path = "sensor_readings.csv"

# --- Writing to a CSV file ---
print("\n--- Writing to CSV File ---")
sensor_data = [
    ['Timestamp', 'Sensor_ID', 'Value', 'Unit'],  # Header row
    ['2025-07-01 10:05:01', 'TEMP_A', 28.5, 'C'],
    ['2025-07-01 10:05:02', 'PRES_B', 101.5, 'kPa'],
    ['2025-07-01 10:05:03', 'FLOW_C', 15.3, 'LPM'],
    ['2025-07-01 10:05:04', 'HUM_D', 65.2, '%RH'],
    ['2025-07-01 10:05:05', 'VOLT_E', 5.02, 'V']
]

try:
    # 'newline=""' avoids extra blank lines on Windows
    with open(csv_file_path, 'w', newline='') as csvfile:
        csv_writer = csv.writer(csvfile)
        csv_writer.writerows(sensor_data)
    print(f"Successfully wrote to '{csv_file_path}'.")
except IOError as e:
    print(f"Error writing CSV file: {e}")

# --- Reading from a CSV file ---
print("\n--- Reading from CSV File ---")
read_data = []
try:
    with open(csv_file_path, 'r', newline='') as csvfile:
        csv_reader = csv.reader(csvfile)
        for row in csv_reader:
            read_data.append(row)
            print(f"Read row: {row}")
    print(f"Successfully read from '{csv_file_path}'. Total rows: {len(read_data)}")
except FileNotFoundError:
    print(f"Error: File '{csv_file_path}' not found.")
except IOError as e:
    print(f"Error reading CSV file: {e}")
```

```python
# --- Processed CSV Data (after reading) ---
print("\n--- Processed CSV Data (after reading) ---")
if read_data:
    headers = read_data[0]
    data_rows = read_data[1:]
    processed_records = []

    for row in data_rows:
        try:
            timestamp, sensor_id, value_str, unit = row
            value = float(value_str)
            processed_records.append({
                'Timestamp': timestamp,
                'Sensor_ID': sensor_id,
                'Value': value,
                'Unit': unit
            })
        except (ValueError, IndexError) as e:
            print(f"Skipping malformed row: {row} - Error: {e}")

    for record in processed_records:
        print(record)
```

1.3 JSON (JavaScript Object Notation) Files

JSON is a lightweight data-interchange format, widely used for data exchange between web applications and services. It maps directly to Python dictionaries and lists.

Conceptual Approach:

Use `json.dump()` to write Python dictionaries/lists to a JSON file and `json.load()` to read JSON data from a file into Python dictionaries/lists.

Code Implementation: JSON File Operations

```python
import json

json_file_path = "device_config.json"

# --- Writing to a JSON file ---
print("\n--- Writing to JSON File ---")
device_configuration = {
    "device_name": "TelemetryUnit_01",
    "version": "1.2.0",
    "sensors": [
        {"id": "TS_A", "type": "temperature", "unit": "Celsius", "threshold":
30.0},
        {"id": "PS_B", "type": "pressure", "unit": "kPa", "threshold": 110.0}
    ],
    "network_settings": {
        "ip_address": "192.168.1.10",
        "port": 8080,
        "protocol": "TCP"
    },
    "logging_enabled": True
}

try:
    # 'indent=4' pretty-prints the JSON for readability
    with open(json_file_path, 'w') as jsonfile:
        json.dump(device_configuration, jsonfile, indent=4)
    print(f"Successfully wrote to '{json_file_path}'.")
except IOError as e:
    print(f"Error writing JSON file: {e}")

# --- Reading from a JSON file ---
print("\n--- Reading from JSON File ---")
loaded_config = {}

try:
    with open(json_file_path, 'r') as jsonfile:
        loaded_config = json.load(jsonfile)
    print(f"Successfully loaded from '{json_file_path}'.")
```

```python
        print("\nLoaded Configuration:")
        print(json.dumps(loaded_config, indent=4))  # Re-pretty-print loaded data

except FileNotFoundError:
    print(f"Error: File '{json_file_path}' not found.")
except json.JSONDecodeError as e:
    print(f"Error decoding JSON: {e}. File might be corrupted.")
except IOError as e:
    print(f"Error reading JSON file: {e}")

# --- Accessing JSON values safely ---
if loaded_config:
    print(f"\nDevice Name: {loaded_config.get('device_name')}")
    print(f"First Sensor ID: {loaded_config['sensors'][0]['id']}")
    print(f"Network Port: {loaded_config['network_settings']['port']}")
```

1.4 Pickle (Serialization/Deserialization)

The **pickle** module implements binary protocols for serializing and de-serializing a Python object structure. "Pickling" is the process of converting a Python object hierarchy into a byte stream, and "unpickling" is the reverse operation. It's Python-specific and not interoperable with other languages.

Conceptual Approach:

Use **pickle.dump()** to write any Python object (not just simple data types) to a binary file and **pickle.load()** to read it back.

Warnings:

- **Security:** Never unpickle data from an untrusted source, as it can execute arbitrary code.
- **Version Compatibility:** Pickled data might not be compatible across different Python versions.

Code Implementation: Pickle Operations

```python
Python
import pickle

pickle_file_path = "complex_data.pkl"

# --- Define a simple custom class for pickling ---
class SensorReading:
    def __init__(self, sensor_id, timestamp, value, unit):
        self.sensor_id = sensor_id
        self.timestamp = timestamp
        self.value = value
        self.unit = unit

    def __repr__(self):
        return (
            f"SensorReading('{self.sensor_id}', '{self.timestamp}', "
            f"{self.value}, '{self.unit}')"
        )

    def __eq__(self, other):
        if not isinstance(other, SensorReading):
            return NotImplemented
        return (
            self.sensor_id == other.sensor_id and
            self.timestamp == other.timestamp and
            self.value == other.value and
            self.unit == other.unit
        )


# --- Writing (pickling) objects to a file ---
print("\n--- Writing (Pickling) Objects ---")
data_to_pickle = [
    SensorReading("TS-101", "2025-07-01T11:00:00", 24.8, "C"),
    SensorReading("PS-202", "2025-07-01T11:00:05", 99.5, "kPa"),
    {"event": "SystemBoot", "time": "2025-07-01T10:00:00", "status": "OK"},
    (1, "error_code_A", [10, 20, 30])
]
```

```python
try:
    with open(pickle_file_path, 'wb') as pklfile:  # 'wb' = write binary
        pickle.dump(data_to_pickle, pklfile)
    print(f"Successfully pickled {len(data_to_pickle)} objects to
'{pickle_file_path}'.")
except IOError as e:
    print(f"Error pickling data: {e}")


# --- Reading (unpickling) objects from a file ---
print("\n--- Reading (Unpickling) Objects ---")
loaded_data = []

try:
    with open(pickle_file_path, 'rb') as pklfile:  # 'rb' = read binary
        loaded_data = pickle.load(pklfile)
    print(f"Successfully unpickled {len(loaded_data)} objects from
'{pickle_file_path}'.")
    for item in loaded_data:
        print(item)
except FileNotFoundError:
    print(f"Error: File '{pickle_file_path}' not found.")
except pickle.UnpicklingError as e:
    print(f"Error unpickling data: {e}. File might be corrupted or
incompatible.")
except IOError as e:
    print(f"Error reading pickle file: {e}")


# --- Verify loaded data type and content ---
if loaded_data:
    print(f"\nType of first loaded item: {type(loaded_data[0])}")
    if isinstance(loaded_data[0], SensorReading):
        print(f"Sensor ID of first item: {loaded_data[0].sensor_id}")
```

2. Context Managers (`with` statement)

The `with` statement in Python is a control flow structure that simplifies resource management by ensuring that resources are properly acquired and released (e.g., files are closed, locks are released), even if errors occur during processing. Objects that support the `with` statement are called context managers.

2.1 The `with` Statement

Conceptual Approach:

The `with` statement guarantees that a resource (like an open file) is correctly handled. When the `with` block is entered, the context manager sets up the resource. When the block is exited (normally or due to an exception), the context manager ensures the resource is properly torn down. This is why `open()` with `with` is the recommended pattern.

Code Implementation:

```python
import random

# --- File handling with `with` (preferred method) ---
print("--- Context Manager: File Handling ---")
file_name_with_context = "context_log.txt"
log_entry_example = "2025-07-01 12:00:00 - Device heartbeat OK\n"

try:
    with open(file_name_with_context, 'w') as f:
        f.write(log_entry_example)
        # Simulate an error after writing
        if random.random() > 0.5:
            raise ValueError("Simulated error during file write operation.")
    print(f"Successfully wrote to '{file_name_with_context}' with context
manager.")
except ValueError as e:
    print(f"Caught expected error: {e}. File will still be closed.")
except IOError as e:
    print(f"Error during file operation: {e}.")
```

```python
# Verify the file is closed by trying to access it again
print("Attempting to read file after context manager block (should be
closed):")
try:
    with open(file_name_with_context, 'r') as f:
        content = f.read()
        print(f"Content: {content.strip()}")
except Exception as e:
    print(f"Error accessing file after close: {e} (This should not happen if
closed correctly).")


# --- Custom Context Manager (Illustrative for Engineering Scenarios) ---
# Imagine a critical resource, like a shared instrument or a locked resource
class DeviceLocker:
    def __init__(self, device_id):
        self.device_id = device_id
        self.is_locked = False

    def __enter__(self):
        """Called when entering the 'with' block."""
        print(f"\nAcquiring lock for device {self.device_id}...")
        # Simulate complex lock acquisition logic
        if random.random() < 0.2:  # 20% chance of failing to acquire
            raise RuntimeError(f"Failed to acquire lock for {self.device_id}.")
        self.is_locked = True
        print(f"Lock acquired for {self.device_id}.")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Called when exiting the 'with' block (normally or due to
exception)."""
        if self.is_locked:
            print(f"Releasing lock for device {self.device_id}...")
            # Simulate complex lock release logic
            self.is_locked = False
            print(f"Lock released for {self.device_id}.")
        else:
            print(f"No lock to release for {self.device_id}.")

        if exc_type:
            print(f"Exception occurred inside with block: {exc_type.__name__}:
```

```
{exc_val}")
        return False  # Re-raise the exception if one occurred


print("\n--- Custom Context Manager: Device Lock ---")
device_id_to_lock = "Spectrometer_01"

try:
    with DeviceLocker(device_id_to_lock) as locker:
        print(f"Working with {locker.device_id} (lock is held)...")
        # Simulate an operation that might cause an error
        if random.random() < 0.3:  # 30% chance of error during operation
            raise ConnectionError(f"Connection lost to {device_id_to_lock}
during operation.")
        print(f"Successfully completed operation with {locker.device_id}.")
except RuntimeError as e:
    print(f"Failed to start operation with {device_id_to_lock}: {e}")
except ConnectionError as e:
    print(f"Operation aborted for {device_id_to_lock} due to error: {e}")
```

3. Filesystem Paths: `pathlib` vs `os.path`

Managing file and directory paths is a common task. Python offers two main ways: the older string-based `os.path` module and the newer object-oriented `pathlib` module.

3.1 `os.path` (String-based)

Conceptual Approach:

`os.path` functions operate on strings representing paths. They are useful for basic path manipulation.

Code Implementation: `os.path` Examples

```python
Python
import os

print("--- os.path Examples ---")

# Get current working directory
current_dir = os.getcwd()
print(f"Current working directory: {current_dir}")

# Joining paths
file_name = "sensor_readings.log"
full_path_os = os.path.join(current_dir, "logs", file_name)
print(f"Joined path (os.path): {full_path_os}")

# Checking existence
print(f"Does current_dir exist? {os.path.exists(current_dir)}")
print(f"Does '{file_name}' exist? {os.path.exists(file_name)}")  # Checks in
current_dir

# Getting components
print(f"Directory name of '{full_path_os}': {os.path.dirname(full_path_os)}")
print(f"Base name of '{full_path_os}': {os.path.basename(full_path_os)}")
print(f"Split extension: {os.path.splitext(file_name)}")
```

3.2 `pathlib` (Object-oriented)

Conceptual Approach:

`pathlib` provides a much more intuitive and powerful way to handle paths by treating them as objects. This allows for chaining methods and using operators for path manipulation, making code cleaner and more readable. It automatically handles platform-specific path separators.

Code Implementation: pathlib Examples

```python
Python
from pathlib import Path

print("\n--- pathlib Examples ---")

# Current working directory as a Path object
current_path_pl = Path.cwd()
print(f"Current Path (Path object): {current_path_pl}")

# Constructing paths using the / operator
log_dir = current_path_pl / "logs"
specific_log_file = log_dir / "system_events.log"
print(f"Constructed file path (pathlib): {specific_log_file}")

# Existence and type checks
print(f"Does '{log_dir}' exist? {log_dir.exists()}")
print(f"Is '{current_path_pl}' a directory? {current_path_pl.is_dir()}")
print(f"Is '{specific_log_file}' a file? {specific_log_file.is_file()}")

# Create a new directory if it doesn't exist
new_logs_folder = Path("my_new_logs")
if not new_logs_folder.exists():
    new_logs_folder.mkdir(parents=True, exist_ok=True)
    print(f"Created directory: {new_logs_folder}")

# File writing and reading using pathlib
sample_data_file = new_logs_folder / "sample_output.txt"
sample_data_file.write_text("Hello from pathlib!\nSecond line of data.")
print(f"Wrote to {sample_data_file}")
print(f"Content of
{sample_data_file.name}:\n{sample_data_file.read_text().strip()}")

# List contents of the new directory
print(f"\nContents of '{new_logs_folder.name}':")
for item in new_logs_folder.iterdir():
    print(f" - {item.name} (Is dir: {item.is_dir()}, Is file:
{item.is_file()})")

# Path components
```

```
print(f"\nParts of '{specific_log_file}':")
print(f"  Name:   {specific_log_file.name}")
print(f"  Stem:   {specific_log_file.stem}")
print(f"  Suffix: {specific_log_file.suffix}")
print(f"  Parent: {specific_log_file.parent}")
print(f"  Parents (all ancestors):")
for parent in specific_log_file.parents:
    print(f"    - {parent}")
```

Comparison: `os.path` vs `pathlib`

Feature	os.path (String-based)	pathlib (Object-oriented)
Path Representation	Strings ('my/file.txt')	Path objects (Path('my') / 'file.txt')
Joining Paths	`os.path.join('dir', 'file.txt')`	Path('dir') / 'file.txt' (using / operator, highly readable)
Existence Check	os.path.exists()	Path.exists() (method on Path object)
Type Check (file/dir)	os.path.isfile(), os.path.isdir()	Path.is_file(), Path.is_dir() (methods)
Creating Dirs	os.makedirs()	Path.mkdir()
Read/Write (basic)	Requires open() (separate steps)	Path.read_text(), Path.write_text() (direct methods for simple text)
Readability	Can be less intuitive with string manipulation.	More intuitive and explicit with object methods and operators.
Chaining Ops	Less amenable to chaining.	Methods can be chained (Path.cwd().parent / 'data'.exists()).

Error Handling	More prone to errors from incorrect string paths.	Path objects handle platform differences automatically; methods raise specific exceptions.
Recommendation	Use for legacy code or when interacting with APIs that strictly require string paths.	Preferred for new code due to its object-oriented nature, readability, and robustness.

4. Unit Testing for Robust Code

Unit testing is a software testing method where individual units or components of a software are tested in isolation. The purpose is to validate that each unit of the software performs as designed.

4.1 Why Unit Testing?

- **Correctness**: Ensures that individual functions or methods work as expected.
- **Reliability**: Increases confidence in the overall stability of the software.
- **Refactoring Safety**: Provides a safety net when modifying existing code; if tests pass, refactoring hasn't broken existing functionality.
- **Documentation**: Tests serve as executable documentation for how functions are supposed to behave.
- **Early Bug Detection**: Catches bugs early in the development cycle, reducing debugging time and cost.
- **Enables Collaboration**: Facilitates teamwork by ensuring consistency and preventing regressions.

4.2 `unittest` Framework (Python Standard Library)

`unittest` is Python's built-in unit testing framework, inspired by JUnit. It's a robust framework suitable for formal test suites.

Conceptual Approach:

Tests are organized into classes that inherit from **unittest.TestCase**. Each test method within these classes should start with **test_**. Assertion methods (e.g., **assertEqual**, **assertTrue**, **assertRaises**) are used to check for expected outcomes. **setUp()** and **tearDown()** methods can be used for test setup and cleanup.

Code Implementation: unittest Example

```Python
import unittest
import math

# --- Function to be tested ---
def calculate_projectile_range(initial_velocity: float, launch_angle_degrees:
float) -> float:
    """
    Calculates the horizontal range of a projectile given initial velocity and
launch angle.
    Formula: Range = (v^2 * sin(2 * angle)) / g
    Assumes ideal projectile motion (no air resistance).
    """
    if initial_velocity < 0 or launch_angle_degrees < 0 or launch_angle_degrees
> 90:
        raise ValueError("Invalid input: velocity and angle must be
non-negative, angle <= 90.")

    g = 9.80665  # Acceleration due to gravity (m/s^2)
    angle_radians = math.radians(launch_angle_degrees)

    range_val = (initial_velocity**2 * math.sin(2 * angle_radians)) / g
    return max(0.0, range_val)  # Avoid tiny floating-point negatives


# --- Unit Tests ---
class TestProjectileCalculations(unittest.TestCase):
    """
    Unit tests for the calculate_projectile_range function.
    """

    def test_zero_velocity(self):
        """Should return 0 for zero initial velocity."""
        self.assertEqual(calculate_projectile_range(0, 45), 0.0)
```

```python
    def test_valid_angle(self):
        """Should calculate correct range for valid input."""
        self.assertAlmostEqual(calculate_projectile_range(10, 45), 10.197,
places=3)
        self.assertAlmostEqual(calculate_projectile_range(20, 30), 35.31,
places=2)

    def test_edge_angles(self):
        """Should return 0 range at 0° and 90° launch angles."""
        self.assertAlmostEqual(calculate_projectile_range(10, 0), 0.0)
        self.assertAlmostEqual(calculate_projectile_range(10, 90), 0.0)

    def test_invalid_input_negative_velocity(self):
        """Should raise ValueError for negative velocity."""
        with self.assertRaises(ValueError):
            calculate_projectile_range(-10, 45)

    def test_invalid_input_angle_too_high(self):
        """Should raise ValueError for angles > 90°."""
        with self.assertRaises(ValueError):
            calculate_projectile_range(10, 91)

    def test_invalid_input_negative_angle(self):
        """Should raise ValueError for negative angles."""
        with self.assertRaises(ValueError):
            calculate_projectile_range(10, -10)


# --- Run Tests ---
if __name__ == "__main__":
    print("\n--- Running unittest Tests ---")
    unittest.main(verbosity=2)  # Change to `exit=False` and set argv in
notebooks
```

© Analog Data

Summary of `unittest` Components

Component	Description
`TestCase`	Base class for creating individual test cases.
`test_*` methods	Methods whose names start with `test_`, these are the actual tests.
`setUp()`	Method run before each test method.
`tearDown()`	Method run after each test method.
`assertEqual()`	Asserts that two values are equal.
`assertTrue()`	Asserts that a condition is true.
`assertRaises()`	Asserts that a specific exception is raised.
`assertAlmostEqual()`	Asserts floats are almost equal (for precision issues).

4.3 `pytest` Framework (External Library)

`pytest` is a popular, powerful, and flexible testing framework that makes writing tests significantly easier and more readable than `unittest`, especially for beginners. It automatically discovers tests and uses simple assert statements instead of verbose `self.assertEqual()`.

Installation: `pip install pytest`

Conceptual Approach:

- Tests are typically just functions (not classes) that start with `test_`.
- Use the standard Python `assert` statement.
- **Fixtures**: Special functions (decorated with `@pytest.fixture`) that provide setup, teardown, and reusable test data for test functions. They are passed as arguments to test functions.

Code Implementation: pytest Example (Conceptual, as it runs via terminal)

```python
Python
import pytest
import math

# --- Function to be tested ---
def calculate_projectile_range(initial_velocity: float, launch_angle_degrees:
float) -> float:
    """
    Calculates the horizontal range of a projectile given initial velocity and
launch angle.
    """
    if initial_velocity < 0 or launch_angle_degrees < 0 or launch_angle_degrees
> 90:
        raise ValueError("Invalid input: velocity and angle must be
non-negative, angle <= 90.")

    g = 9.80665
    angle_radians = math.radians(launch_angle_degrees)
    range_val = (initial_velocity**2 * math.sin(2 * angle_radians)) / g
    return max(0.0, range_val)


# --- Fixtures ---
@pytest.fixture
def default_velocity():
    """Provides a default velocity for tests."""
    return 10.0

@pytest.fixture
def default_gravity():
    """Returns gravity constant for manual calculations (not used directly in
function)."""
    return 9.80665


# --- Individual Tests ---
def test_zero_velocity():
    assert calculate_projectile_range(0, 45) == 0.0

def test_valid_angle(default_velocity, default_gravity):
```

```python
    expected_range = (default_velocity**2 * math.sin(2 * math.radians(45))) /
default_gravity
    assert calculate_projectile_range(default_velocity, 45) ==
pytest.approx(expected_range)

def test_negative_velocity_raises():
    with pytest.raises(ValueError, match="Invalid input"):
        calculate_projectile_range(-10, 45)

def test_angle_too_high_raises():
    with pytest.raises(ValueError, match="Invalid input"):
        calculate_projectile_range(10, 100)

def test_negative_angle_raises():
    with pytest.raises(ValueError, match="Invalid input"):
        calculate_projectile_range(10, -5)


# --- Parametrized Testing ---
@pytest.mark.parametrize("velocity, angle, expected", [
    (10, 45, 10.197),
    (20, 30, 35.31),
    (5, 0, 0.0),
    (5, 90, 0.0),
])
def test_projectile_range_cases(velocity, angle, expected):
    assert calculate_projectile_range(velocity, angle) ==
pytest.approx(expected, abs=0.01)
```

Run Instructions (Terminal):

```shell
Shell
pytest test_projectile.py
# Or run all tests in the current directory
pytest
```

Summary: `unittest` vs `pytest`

Feature	unittest (Standard Library)	pytest (External Library)
Test Discovery	Based on `TestCase` classes and `test_*` methods.	Automatically finds `test_.*py` files and `test_*` functions/methods.
Assertions	`self.assertEqual()`, `self.assertTrue()`, etc.	Standard Python `assert` statement.
Fixtures/Setup	`setUp()`, `tearDown()` methods (per test/class).	`@pytest.fixture` functions (flexible, reusable, dependency injection).
Parametrization	Manual loops or external libraries.	`@pytest.mark.parametrize` (very powerful and concise).
Extensibility	Plugins available, but less common.	Rich plugin ecosystem (`pytest-cov`, `pytest-html`).
Readability	More verbose syntax.	Often more concise and readable.
Learning Curve	Slightly steeper for complex setups.	Flatter, intuitive for basic tests.
Installation	Built-in.	`pip install pytest`.
Recommendation	Good for formal, strict test structures.	Recommended for most new Python projects due to ease of use and powerful features.

4.4 Mocking for Tests

Conceptual Approach:

Mocking is a technique used in unit testing to isolate the code being tested from its dependencies (e.g., external services, databases, file systems, hardware devices). Instead of using real dependencies, you replace them with "mock" objects that simulate the behavior of the real ones. This makes tests faster, more reliable, and independent of external factors.

The **unittest.mock** module (built into Python 3.3+) provides powerful tools for creating and managing mock objects. The most common tool is patch.

Code Implementation: Mocking File I/O (Conceptual for real files)

```python
from unittest.mock import patch, mock_open

# --- Function that interacts with a file (to be tested) ---
def read_device_status_from_file(status_file_path: str) -> str:
    """Reads device status from a specified file."""
    try:
        with open(status_file_path, 'r') as f:
            status = f.readline().strip()
            return status
    except FileNotFoundError:
        return "ERROR: Status file not found."
    except Exception as e:
        return f"ERROR: Could not read status: {e}"

# --- Test code using mock_open to simulate file reading ---
# This part would typically be in a test file (e.g., test_device_reader.py)

print("--- Mocking File I/O with unittest.mock.patch and mock_open ---")

# Scenario 1: Simulate successful file read
mock_file_content = "DEVICE_ACTIVE\n"
m = mock_open(read_data=mock_file_content)
```

```python
# Replace builtins.open with our mock for this context
with patch('builtins.open', m):
    status = read_device_status_from_file("any_path_does_not_matter.txt")
    print(f"Simulated read status (Active): {status}")
    m.assert_called_once_with("any_path_does_not_matter.txt", 'r')

# Scenario 2: Simulate file not found
m_not_found = mock_open()
m_not_found.side_effect = FileNotFoundError

with patch('builtins.open', m_not_found):
    status_not_found = read_device_status_from_file("non_existent_status.txt")
    print(f"Simulated read status (Not Found): {status_not_found}")
    m_not_found.assert_called_once_with("non_existent_status.txt", 'r')

# --- Mocking Serial Input (Conceptual) ---
# When testing code that communicates with hardware via a serial port (e.g.,
pySerial),
# you would mock the pySerial.Serial class or its methods.

# Example of a function that might use a serial port
# def read_from_serial_device(port, baudrate):
#     import serial
#     ser = serial.Serial(port, baudrate, timeout=1)
#     line = ser.readline().decode('utf-8').strip()
#     ser.close()
#     return line

# Conceptual mocking
# with patch('serial.Serial') as mock_serial_class:
#     mock_instance = mock_serial_class.return_value
#     mock_instance.readline.return_value = b'SENSOR_DATA:25.5\r\n'
#     result = read_from_serial_device('/dev/ttyUSB0', 9600)
#     assert result == 'SENSOR_DATA:25.5'
#     mock_serial_class.assert_called_once_with('/dev/ttyUSB0', 9600,
timeout=1)
#     mock_instance.readline.assert_called_once()
#     mock_instance.close.assert_called_once()

print("\n--- Mocking Serial Input (Conceptual) ---")
print("To test functions interacting with hardware (e.g., serial ports):")
print("1. Use `unittest.mock.patch` to replace the `serial.Serial` class (or
```

```
specific methods).")
print("2. Configure the mock object's return values for methods like
`readline()` or `write()`.")
print("3. Assert that the mock methods were called correctly.")
print("This isolates your code from actual hardware, making tests fast and
reliable.")
```

5. Debugging & Tracebacks

Debugging is the process of finding and fixing errors or bugs in your code. A **traceback** is a report that provides a summary of the calls made in your code before an exception occurred, helping you locate the source of the error.

5.1 Understanding Tracebacks

Conceptual Approach:

When an unhandled exception occurs, Python prints a traceback. This message is your primary tool for understanding what went wrong, where it went wrong, and how your code got to that point.

Key Parts of a Traceback:

- `Traceback (most recent call last):`: Indicates the start of the traceback.
- **File and Line Number**: Points to the exact location (file, line number) where the error occurred. This is the most crucial part.
- **Function Name**: The name of the function or method where the error happened.
- **Error Type and Message**: The type of exception (e.g., `ValueError`, `IndexError`) and a brief explanatory message.
- **Call Stack**: A list of function calls that led to the error, from the outermost call to the innermost.

Code Implementation: Common Tracebacks

```python
Python
print("--- Understanding Tracebacks ---")

# Example 1: NameError
def function_a():
    function_b()

def function_b():
    print(unknown_variable)  # NameError here

try:
    function_a()
except NameError as e:
    print(f"\nCaught (simulated) NameError: {e}")

# Example 2: IndexError in a nested loop
def process_matrix(matrix):
    for row_idx, row in enumerate(matrix):
        for col_idx in range(len(row) + 1):  # Intentional error: accessing out
of bounds
            item = row[col_idx]
            print(f"Item: {item}")

try:
    my_matrix = [[1, 2], [3, 4]]
    process_matrix(my_matrix)
except IndexError as e:
    print(f"\nCaught (simulated) IndexError: {e}")

# Example 3: TypeError in function call
def operate_device(device_obj, command):
    device_obj.send_command(command)  # Assume send_command expects string

class DummyDevice:
    def send_command(self, cmd_str):
        if not isinstance(cmd_str, str):
            raise TypeError("Command must be a string!")
        print(f"Command '{cmd_str}' sent to device.")
```

```python
try:
    device = DummyDevice()
    operate_device(device, 123)  # Pass int instead of string
except TypeError as e:
    print(f"\nCaught (simulated) TypeError: {e}")
```

5.2 Debugging with pdb (Python Debugger)

pdb is Python's built-in command-line debugger. It allows you to pause program execution, inspect variables, step through code line by line, and evaluate expressions.

Conceptual Approach:

- You can start **pdb** by inserting **breakpoint()** (Python 3.7+) or **import pdb; pdb.set_trace()** at the point where you want to pause execution.
- When the program hits the breakpoint, it enters an interactive **(Pdb)** prompt.
- You then use **pdb** commands to control execution and inspect the state.

Common pdb Commands

Command	Description
l	List source code around current line.
n	Next line. Steps over function calls.
s	Step into. Steps into function calls.
c	Continue execution until the next breakpoint or end of program.
p <expr>	Print the value of an expression.
pp <expr>	Pretty-print the value of an expression.
b <line_num>	Set a breakpoint at a specific line number.
cl	Clear all breakpoints.

q	Quit the debugger (and terminate the program).
h	Help on commands.
a	Print arguments of the current function.
w	Where on the call stack you are.

Code Implementation: pdb Example (Interactive, requires careful execution)

```python
# To run this, uncomment the `breakpoint()` line.
# When it hits `(Pdb)`, you will type commands in the input area below cell.

def apply_calibration_factor(raw_value: float, factor: float) -> float:
    """Applies a calibration factor to a raw sensor value."""
    # breakpoint()  # Uncomment this line to start debugging here
    calibrated = raw_value * factor
    return calibrated

def process_sensor_stream_pdb(sensor_readings: list, global_factor: float):
    """Processes a stream of sensor readings with a global calibration."""
    processed_results = []
    for i, reading in enumerate(sensor_readings):
        print(f"Processing reading {i}: {reading}")
        adjusted_reading = apply_calibration_factor(reading, global_factor)
        processed_results.append(adjusted_reading)
    return processed_results

# Example Usage to trigger pdb
# print("\n--- Debugging with pdb ---")
# data_points = [10.0, 15.0, 20.0]
# factor = 1.2
# final_data = process_sensor_stream_pdb(data_points, factor)
# print(f"Final processed data: {final_data}")

# When `breakpoint()` is hit:
# (Pdb) l              # list code
# (Pdb) p raw_value    # print raw_value
# (Pdb) n              # next line
# (Pdb) c              # continue execution
```

Note for Jupyter/IDE Users:

While `pdb` is powerful, most IDEs (Spyder, VS Code, PyCharm) offer a much more user-friendly graphical debugger. You can set breakpoints by clicking in the margin next to the line number, inspect variables in a dedicated pane, and use step-over, step-into, and step-out buttons. For everyday development, the IDE debugger is generally preferred.

6. Key Takeaways

This session has equipped you with essential practical skills for building professional and reliable Python applications in engineering and scientific contexts.

- **File I/O Mastery**: Learned how to read and write various data formats (text, CSV, JSON, Pickle), understanding their appropriate use cases and the best practices for handling them.
- **Context Managers**: Mastered the `with` statement for automatic and safe resource management, preventing common issues like unclosed files or forgotten resource releases.
- **Modern Path Handling**: Gained proficiency with the `pathlib` module for object-oriented filesystem path manipulation, offering a cleaner and more robust alternative to `os.path`.
- **Unit Testing Principles**: Understood the critical importance of unit testing for code correctness, reliability, and maintainability, especially in complex systems.
- `unittest` & `pytest`: Explored both Python's built-in `unittest` framework and the powerful external `pytest` framework, learning how to write effective tests, use fixtures, and leverage features like parametrization.
- **Mocking**: Grasped the concept of mocking to isolate code under test from external dependencies (like files or hardware), leading to faster and more reliable tests.
- **Debugging Essentials**: Learned to effectively read and interpret tracebacks to diagnose errors and gained a foundational understanding of interactive debugging with pdb and the benefits of IDE debuggers.

These practical skills are indispensable for any engineer or scientist who develops Python software, enabling you to deliver higher quality, more stable, and easier-to-maintain solutions.