Day 3 Session 3:
Exception Handling & Robust Code - Deep Dive

This notebook is crucial for developing reliable and resilient Python applications. We will delve into Python's exception handling mechanisms, learning how to gracefully manage errors, validate assumptions, and build robust code that can withstand unexpected conditions in engineering and scientific systems.

1. Introduction to Exception Handling & Robust Code

In the real world of engineering and scientific computing, things don't always go as planned. Sensors can fail, networks can drop, user input can be invalid, and calculations can lead to impossible results (like division by zero). Without proper handling, these "exceptional" situations can cause your program to crash unexpectedly, leading to data loss, system downtime, or even dangerous operational errors.

Exception handling is a programming mechanism that allows you to manage these runtime errors gracefully. Instead of crashing, your program can "catch" the error, perform cleanup, log the issue, and potentially recover or exit in a controlled manner.

Robust code is code designed to be resilient to errors, unexpected inputs, and failures. It is stable, handles edge cases, and provides clear feedback when something goes wrong.

2. Basic Exception Handling: `try`, `except`, `else`, `finally`

The core of Python's exception handling is the `try...except` block. It allows you to "try" to execute a block of code and "catch" specific errors (exceptions) that might occur.

2.1 The `try...except` Block

The `try` block contains the code that might raise an exception. The `except` block specifies how to handle a particular exception if it occurs in the `try` block.

Conceptual Approach:

Wrap the potentially problematic code in a `try` block. If an error matching an `except` clause occurs, the `try` block immediately terminates, and the code inside the matching `except` block is executed. If no exception occurs, the `except` block is skipped.

Example 1: Division by Zero

```python
print("--- Basic try-except: Division by Zero ---")

numerator = 10

# Valid case
denominator1 = 2
try:
    result = numerator / denominator1
    print(f"Result of {numerator} / {denominator1} = {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")

# Invalid case
denominator2 = 0
try:
    result = numerator / denominator2
    print(f"Result of {numerator} / {denominator2} = {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
```

Example 2: Invalid Sensor Data Conversion

```python
print("\n--- Basic try-except: Invalid Data Conversion ---")

raw_sensor_value_good = "25.7"
raw_sensor_value_bad = "twenty five"
```

```
# Case 1: Valid input
try:
    # Attempt to convert a string to a float
    reading = float(raw_sensor_value_good)
    print(f"Successfully converted '{raw_sensor_value_good}' to {reading:.2f}")
except ValueError:
    print(f"Error: Could not convert '{raw_sensor_value_good}' to a number.
Invalid format.")

# Case 2: Invalid input
try:
    reading = float(raw_sensor_value_bad)
    print(f"Successfully converted '{raw_sensor_value_bad}' to {reading:.2f}")
except ValueError:
    print(f"Error: Could not convert '{raw_sensor_value_bad}' to a number.
Invalid format.")
```

2.2 Handling Multiple Specific Exceptions

You can have multiple `except` blocks to handle different types of exceptions. Python will execute the first `except` block whose exception type matches the error that occurred, and then skip the remaining `except` blocks. You can also group exceptions into a single `except` clause using a tuple.

Conceptual Approach:

Order matters. More specific exceptions should generally be caught before more general ones.

Code Implementation:

Python

```python
print("\n--- Handling Multiple Specific Exceptions ---")

def process_instrument_data(data_record: str):
    """
    Processes an instrument data record, handling common parsing and conversion
errors.
```

```
    Expected format: "SENSOR_ID:VALUE:UNIT"
    """
    try:
        parts = data_record.split(':')
        sensor_id = parts[0].strip()
        value = float(parts[1].strip())
        unit = parts[2].strip()
        print(f"Processed {sensor_id}: Reading = {value:.2f} {unit}")
    except ValueError:
        print(f"Error: Invalid numeric value in '{data_record}'.")
    except IndexError:  # Raised if split does not produce enough parts (e.g.,
"ID-VALUE")
        print(f"Error: Malformed data string '{data_record}'. Expected
'ID:VALUE:UNIT'.")
    except TypeError as e:  # Example: If data_record is not a string (e.g.,
None)
        print(f"Error: Invalid data type for input: {e}")
    except Exception as e:  # Catch any other unexpected exception (general
fallback)
        print(f"An unexpected error occurred: {type(e).__name__} - {e}")

# Test cases
process_instrument_data("TEMP_A01:25.5:C")          # Valid
process_instrument_data("PRES_B02:abc:kPa")         # ValueError
process_instrument_data("FLOW_C03-12.3-LPM")        # IndexError
process_instrument_data("HUM_D04:101.2:RH:extra")   # Still processes first 3
fields

try:
    # This will cause a TypeError because None does not have a .split() method
    process_instrument_data(None)
except Exception as e:
    print(f"Caught top-level error when passing None: {type(e).__name__} -
{e}")

print("\n--- Catching Multiple Exceptions in One Block ---")

def safely_access_data(container, index):
    try:
        value = container[index]
        converted = float(value)
        print(f"Accessed and converted: {converted}")
    except (IndexError, ValueError) as e:  # Catch either if they occur
```

```
        print(f"Data access or conversion error: {type(e).__name__} - {e}")

# Test cases
safely_access_data([10, '20.5', 30], 0)  # Valid
safely_access_data([10, '20.5', 30], 3)  # IndexError
safely_access_data([10, 'abc', 30], 1)   # ValueError
```

2.3 Catching All Exceptions (`except Exception as e`)

You can use a general `except Exception as e` block to catch any exception that inherits from `Exception`. While convenient, it's generally **discouraged** as the primary error handling mechanism because it can mask unexpected bugs and make debugging harder. It's best used as a last resort in a multi-except block or for logging unknown issues before re-raising.

```python
print("\n--- Catching All Exceptions (Use with Caution) ---")

def very_risky_operation(data_list, divisor):
    try:
        # This might cause ZeroDivisionError
        result = 100 / divisor
        # This might cause IndexError
        value = data_list[5]
        print(f"Result: {result}, Value: {value}")
    except ZeroDivisionError:
        print("Specific error: Division by zero handled here.")
    except Exception as e:  # Catches IndexError and any other unexpected
Exception
        print(f"General error occurred in risky operation: {type(e).__name__} -
{e}")

# Test cases
very_risky_operation([1, 2, 3], 0)        # Triggers ZeroDivisionError
very_risky_operation([1, 2, 3], 10)       # Triggers IndexError
very_risky_operation("not a list", 5)    # Triggers TypeError
```

2.4 The `else` Block

The `else` block (used with `try...except`) is executed only if the code in the `try` block completes without raising any exceptions. It's useful for placing code that should only run if the `try` block was successful and free of errors.

Python
```python
print("\n--- The else Block (Executed on Success) ---")

def read_and_process_config(config_name):
    config_data = None
    try:
        # Simulate reading a configuration file
        if config_name == "app_config.json":
            config_data = {"version": "1.2", "mode": "AUTO", "threshold": 50.5}
            print(f"Successfully loaded '{config_name}'.")
        elif config_name == "missing_config.json":
            raise FileNotFoundError(f"Configuration file '{config_name}' not
found.")
        else:
            raise ValueError("Unsupported configuration name.")
    except FileNotFoundError:
        print(f"Error: '{config_name}' does not exist. Using default
settings.")
        config_data = {"version": "1.0", "mode": "MANUAL", "threshold": 25.0}
    except ValueError as e:
        print(f"Error processing config: {e}. Using safe defaults.")
        config_data = {"version": "1.0", "mode": "MANUAL", "threshold": 25.0}
    else:
        # This block runs ONLY if NO exception was raised in the try block
        print(f"Configuration loaded successfully. Version:
{config_data['version']}")
        print(f"Mode: {config_data['mode']}, Threshold:
{config_data['threshold']}")
        if config_data['mode'] == "AUTO":
            print("Auto mode enabled. System will operate autonomously.")

# Test calls
read_and_process_config("app_config.json")
print("-" * 30)
```

```
read_and_process_config("missing_config.json")
print("-" * 30)

read_and_process_config("invalid_name.xml")
```

2.5 The `finally` Block

The **`finally`** block is always executed, regardless of whether an exception occurred in the **`try`** block, an **`except`** block handled it, or the **`else`** block ran. It's typically used for cleanup operations that must happen (e.g., closing files, releasing hardware resources, disconnecting network connections) to prevent resource leaks.

Python
```python
import random

print("\n--- The finally Block (Guaranteed Execution) ---")

def manage_device_communication(device_id: str, simulate_error: bool = False,
skip_disconnect: bool = False):
    print(f"Attempting to communicate with device {device_id}...")
    device_connection = None  # Represents a connection object

    try:
        # Simulate establishing a connection
        print(f"  Establishing connection to {device_id}...")
        if random.random() < 0.1 or simulate_error:  # 10% chance of connection
error, or forced
            raise ConnectionError(f"Failed to connect to {device_id}.")

        device_connection = {"status": "connected", "id": device_id}
        print(f"  Connection to {device_id} successful.")

        # Simulate data transfer (might fail)
        if random.random() < 0.2:  # 20% chance of data error
            raise ValueError("Data integrity compromised during transfer.")
        print(f"  Data transfer with {device_id} successful.")

    except ConnectionError as e:
        print(f"  Caught: Connection error with {device_id}: {e}")
    except ValueError as e:
        print(f"  Caught: Data transfer error with {device_id}: {e}")
    except Exception as e:
        print(f"  Caught: An unexpected error occurred: {type(e).__name__} -
{e}")
    finally:
        # This block always executes, crucial for cleanup
        if device_connection and not skip_disconnect:
            print(f"  [FINALLY] Disconnecting from device {device_id}.")
            # In a real system: device_connection.disconnect()
        elif skip_disconnect:
            print(f"  [FINALLY] Explicitly skipping disconnect for
{device_id}.")
        else:
            print(f"  [FINALLY] No active connection for {device_id} to
disconnect.")

    print(f"Finished communication attempt with {device_id}.\n")

# Scenario 1: Successful connection and operation
manage_device_communication("Sensor_X01")

# Scenario 2: Connection error
manage_device_communication("Sensor_X02", simulate_error=True)

# Scenario 3: Data transfer error (but connection closes)
manage_device_communication("Sensor_X03")  # Let it randomly fail or pass

# Scenario 4: No disconnect for testing
manage_device_communication("Sensor_X04", skip_disconnect=True)
```

Summary of `try-except-else-finally` Flow

Block	Purpose	Execution Condition
`try`	Contains code that might raise an exception.	Always executed.
`except`	Catches and handles specific exceptions.	Executed if an exception matching its type occurs in `try`.
`else`	Contains code that runs if `try` succeeds.	Executed only if no exception is raised in `try`.
`finally`	Contains cleanup code.	Always executed, regardless of exceptions or return statements.

3. Built-in Exceptions and Raising Custom Exceptions

Python has a rich hierarchy of built-in exceptions for common error conditions. You can also define and raise your own custom exceptions to represent specific error scenarios in your application.

3.1 Common Built-in Exceptions

Python's built-in exceptions form a class hierarchy, with `BaseException` at the root, and `Exception` as the base for most common runtime errors you should handle.

Table of Common Built-in Exceptions (and When They Occur)

Exception Type	Description	Example Trigger
`SyntaxError`	Invalid Python syntax.	`if x = 10:`
`IndentationError`	Incorrect indentation.	`print("hi")` (when not expected)

`NameError`	Using an undefined variable/name.	`print(undefined_var)`
`TypeError`	Operation on an inappropriate type.	`len(123)`, `'a' + 5`
`ValueError`	Correct type, but inappropriate value.	`int("hello")`, `math.sqrt(-1)` (for real numbers)
`IndexError`	List/tuple index out of range.	`my_list` (for len=5 list)
`KeyError`	Dictionary key not found.	`my_dict['non_existent_key']`
`AttributeError`	Attempting to access non-existent attribute/method.	`my_object.non_existent_method()`
`FileNotFoundError`	File/directory not found.	`open('non_existent.txt')`
`ZeroDivisionError`	Division or modulo by zero.	`10 / 0`
`IOError`	General I/O operation failure (e.g., disk full, permissions).	`open('file.txt', 'w')` (no permission)
`OSError`	Operating system related errors (e.g., file system, process).	(Often base for `FileNotFoundError`, etc.)
`MemoryError`	Program runs out of memory.	Creating very large data structures.
`RecursionError`	Function calls itself too many times (infinite recursion).	Undefined base case in recursive function.

3.2 Raising Exceptions (`raise`)

You can explicitly raise an exception in your code when an error condition occurs. This is useful for signaling that a function cannot complete its task due to invalid input, an unexpected state, or a critical failure.

Conceptual Approach:

Use the `raise` keyword followed by an exception class (and optionally an error message).

Example:

Python
```python
print("\n--- Raising Exceptions ---")

def check_positive(number):
    if number < 0:
        raise ValueError("Number must be positive.")
    return number

try:
    check_positive(10)
    check_positive(-5)  # This will raise ValueError
except ValueError as e:
    print(f"Caught error: {e}")
```

3.3 Creating Custom Exception Classes

For application-specific error conditions, it's good practice to define your own custom exception classes. This makes your code more readable, allows for more specific error handling, and makes it easier to categorize and diagnose issues.

Conceptual Approach:

Custom exceptions should inherit from a built-in `Exception` class (usually `Exception` itself, or a more specific one like `ValueError` if your custom exception is a type of value error).

Code Implementation:

Python

```python
# Define custom exception classes for a telemetry system
class TelemetryError(Exception):
    """Base exception for all telemetry-related errors."""
    pass

class SensorReadError(TelemetryError):
    """Raised when a sensor reading fails or is corrupted."""
    def __init__(self, sensor_id, message="Failed to read from sensor"):
        self.sensor_id = sensor_id
        self.message = f"{message} (Sensor ID: {sensor_id})"
        super().__init__(self.message)

class DataValidationError(TelemetryError):
    """Raised when telemetry data fails validation rules."""
    def __init__(self, field_name, value, expected_range, message="Data
validation failed"):
        self.field_name = field_name
        self.value = value
        self.expected_range = expected_range
        self.message = (
            f"{message}: Field '{field_name}' with value '{value}' "
            f"is not within expected range {expected_range}."
        )
        super().__init__(self.message)


def process_telemetry_packet(packet_data: dict):
    """
    Simulates processing a telemetry packet, raising custom exceptions.
    """
    print(f"\nProcessing packet: {packet_data.get('id', 'N/A')}...")

    sensor_id = packet_data.get("sensor_id")
    temperature = packet_data.get("temperature")
    pressure = packet_data.get("pressure")

    if sensor_id is None:
        raise SensorReadError("UNKNOWN", "Sensor ID missing in packet.")
    if not isinstance(temperature, (int, float)):
        raise SensorReadError(sensor_id, f"Invalid temperature value:
{temperature}.")

    if not (0 <= temperature <= 100):  # Assuming 0-100 °C is expected
        raise DataValidationError("temperature", temperature, (0, 100))

    if pressure is not None and not isinstance(pressure, (int, float)):
        raise DataValidationError("pressure", pressure, "numeric value")

    print(
        f"  Packet {sensor_id} processed: "
        f"Temp = {temperature:.2f}, Pressure = {pressure if pressure is not
None else 'N/A'}"
    )


# --- Test cases for custom exceptions ---
print("--- Using Custom Exception Classes ---")

# 1. Valid packet
try:
    process_telemetry_packet({
        "id": 1,
        "sensor_id": "TM-001",
        "temperature": 25.5,
        "pressure": 101.2
    })
except TelemetryError as e:
    print(f"Caught TelemetryError: {e}")

# 2. Missing sensor ID
try:
    process_telemetry_packet({
        "id": 2,
        "temperature": 30.0,
        "pressure": 99.0
    })
except SensorReadError as e:
    print(f"Caught SensorReadError: {e}")
except TelemetryError as e:
    print(f"Caught general TelemetryError: {e}")
```

```
# 3. Invalid temperature value (type error)
try:
    process_telemetry_packet({
        "id": 3,
        "sensor_id": "TM-002",
        "temperature": "hot",
        "pressure": 100.0
    })
except SensorReadError as e:
    print(f"Caught SensorReadError: {e}")

# 4. Temperature out of range (value error)
try:
    process_telemetry_packet({
        "id": 4,
        "sensor_id": "TM-003",
        "temperature": 150.0,
        "pressure": 102.0
    })
except DataValidationError as e:
    print(f"Caught DataValidationError: {e}")
```

Summary: Raising and Custom Exceptions

Concept	Description	Benefit
`raise`	Keyword used to explicitly trigger an exception.	Signals critical errors, forces calling code to handle.
Built-in Exceptions	Python's standard error types (e.g., `ValueError`, `FileNotFoundError`).	Cover common scenarios, widely understood.
Custom Exceptions	User-defined classes that inherit from `Exception` (or a subclass).	Provide specific, meaningful error types for your application domain.

4. Assertion-Based Debugging (`assert`)

The `assert` statement is a debugging aid that checks if a condition is `True`. If the condition is `False`, it raises an `AssertionError`. Assertions are primarily used to test assumptions that the programmer believes to be true during development.

Conceptual Approach:

Use `assert condition, "Error message"` to state assumptions that should always be true at that point in the code. If an assertion fails, it indicates a bug in the program logic, not an expected error that needs graceful handling for the end-user.

Example 1:

```python
print("--- Assertion-Based Debugging ---")

def divide_safe(a, b):
    # Assert that b is not zero. If this assertion fails, it indicates a programming error.
    assert b != 0, "Denominator cannot be zero. This is a programming bug."
    return a / b

# Valid division
print(f"10 / 2 = {divide_safe(10, 2)}")

# Division by zero triggers an AssertionError
try:
    print(f"10 / 0 = {divide_safe(10, 0)}")
except AssertionError as e:
    print(f"Caught AssertionError: {e}")
```

Example 2: Asserting Sensor Data Constraints

Python
```python
def process_sensor_data_assert(data_packet: dict):
    """
    Processes a sensor data packet, using assertions for internal sanity
checks.
    Assumes incoming data 'value' is valid for processing.
    """
    print(f"\nProcessing sensor data with assertions: {data_packet.get('id',
'N/A')}...")

    # Assert that critical keys exist before proceeding
    assert "id" in data_packet, "Data packet missing 'id' key."
    assert "value" in data_packet, "Data packet missing 'value' key."
    assert "unit" in data_packet, "Data packet missing 'unit' key."

    sensor_id = data_packet["id"]
    value = data_packet["value"]
    unit = data_packet["unit"]

    # Assert value type and range before calculations
    assert isinstance(value, (int, float)), f"Value for {sensor_id} is not
numeric: {value}"
    assert 0 <= value <= 1000, f"Value {value} for {sensor_id} is outside
expected processing range (0-1000)."

    # If all assertions pass, proceed with processing
    processed_value = value * 0.1  # Example transformation
    print(f"  Processed {sensor_id}: {processed_value:.2f} {unit} (original
{value})")
    return processed_value


print("--- Asserting Sensor Data Constraints ---")

# Valid data
process_sensor_data_assert({"id": "S1", "value": 50, "unit": "C"})

# Missing key
try:
    process_sensor_data_assert({"value": 60, "unit": "kPa"})
```

```
except AssertionError as e:
    print(f"Caught Assertion Error: {e}")

# Invalid value type
try:
    process_sensor_data_assert({"id": "S3", "value": "xyz", "unit": "V"})
except AssertionError as e:
    print(f"Caught Assertion Error: {e}")

# Value out of *assertion* range (assumed internal processing range)
try:
    process_sensor_data_assert({"id": "S4", "value": -10, "unit": "A"})
except AssertionError as e:
    print(f"Caught Assertion Error: {e}")
```

assert vs. raise (Exceptions)

Feature	assert	raise (for general exceptions)
Purpose	Debugging, internal sanity checks, programmer assumptions.	Error handling, signaling expected (but problematic) conditions.
When to Use	For conditions that should never occur if the code is correct.	For expected runtime problems that the program might recover from or needs to inform the user about.
Behavior	Raises `AssertionError`. Can be disabled with `python -O`.	Raises a specified exception type. Cannot be disabled.
Audience	Developers/Testers.	Developers and potentially end-users.
Outcome	Indicates a bug in code logic.	Indicates a problem in data, environment, or external system.

5. Logging vs. Exception Throwing

Choosing between logging an issue and raising an exception is a critical design decision in robust applications.

5.1 Logging

Conceptual Approach:

Logging is used for recording events, status messages, and non-critical issues that don't necessarily disrupt the program's flow or require immediate intervention. It provides a historical record for monitoring, debugging, and auditing.

When to Log:

- Informational messages (e.g., "System started").
- Warnings (e.g., "Optional configuration not found, using default").
- Debugging information (e.g., variable values at certain points).
- Non-fatal errors that allow the program to continue (e.g., a single bad record in a batch processing).

Code Implementation:

Python
```python
import logging
import random

# Configure logging (usually done once at application start)
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

# You can also set specific levels for different handlers or loggers
# logging.getLogger().setLevel(logging.DEBUG)  # For more verbose output

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

def process_data_entry(entry_id: str, value: float):
    if value < 0:
        logging.warning(
            f"Data entry {entry_id} has negative value: {value}. "
            "Processing anyway (might lead to incorrect results)."
        )
    elif value > 1000:
        logging.error(
            f"Data entry {entry_id} has extremely high value: {value}. "
            "Skipping this entry."
        )
        return  # Skip processing this problematic entry
    else:
        logging.info(
            f"Processing data entry {entry_id} with value {value:.2f}."
        )

    # Simulate processing
    processed_result = value * 1.5
    logging.debug(
        f"Intermediate result for {entry_id}: {processed_result:.2f}"
    )  # Only shows if level is DEBUG
    return processed_result

# Test cases
process_data_entry("SENSOR_001", 50.0)     # INFO
process_data_entry("SENSOR_002", -10.0)    # WARNING
process_data_entry("SENSOR_003", 1200.0)   # ERROR (skipped)
process_data_entry("SENSOR_004", 250.0)    # INFO
```

5.2 Exception Throwing (Raising)

Conceptual Approach:

Raising an exception is used for critical error conditions that prevent the function from completing its intended task. It forces the calling code to deal with the problem. It interrupts the normal flow of execution.

When to Raise an Exception:

- Invalid arguments that make the function impossible to execute correctly.
- Unrecoverable external resource issues (e.g., database down, critical file missing).
- Violations of fundamental assumptions or contracts that indicate a severe problem.
- Situations where the caller needs to explicitly know about and handle the failure.

Code Implementation (reusing custom exceptions):

```python
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')


# Reusing custom exceptions from section 3.3
class TelemetryError(Exception):
    pass


class SensorReadError(TelemetryError):
    def __init__(self, sensor_id, message="Failed to read from sensor"):
        self.sensor_id = sensor_id
        self.message = f"{message} (Sensor ID: {sensor_id})"
        super().__init__(self.message)


class DataValidationError(TelemetryError):
    def __init__(self, field_name, value, expected_range, message="Data
validation failed"):
        self.field_name = field_name
        self.value = value
        self.expected_range = expected_range
        self.message = (
            f"{message}: Field '{field_name}' with value '{value}' "
            f"is not within expected range {expected_range}."
        )
        super().__init__(self.message)
```

```python
def get_calibrated_sensor_reading(raw_value: float, calibration_factor: float)
-> float:
    if raw_value < 0:
        raise ValueError(f"Raw sensor value cannot be negative: {raw_value}")
    if calibration_factor <= 0:
        raise ZeroDivisionError("Calibration factor cannot be zero or
negative.")
    return raw_value * calibration_factor


def simulate_device_operation(device_name: str, raw_temp: float, cal_factor:
float):
    print(f"\nSimulating operation for device {device_name}...")

    try:
        # Step 1: Get calibrated reading
        calibrated_temp = get_calibrated_sensor_reading(raw_temp, cal_factor)
        logging.info(f"Device {device_name}: Calibrated temperature
{calibrated_temp:.2f}°C.")

        # Step 2: Simulate power draw based on temperature
        if calibrated_temp > 50:
            logging.warning(
                f"Device {device_name}: High temperature detected
({calibrated_temp:.2f}°C). Power draw will be elevated."
            )
            power_draw = calibrated_temp * 5.0
        elif calibrated_temp < 10:
            logging.warning(
                f"Device {device_name}: Low temperature detected
({calibrated_temp:.2f}°C). Might affect efficiency."
            )
            power_draw = calibrated_temp * 2.0
        else:
            power_draw = calibrated_temp * 3.0

        # Step 3: Validate power draw
        if power_draw > 500:
            raise TelemetryError(
                f"Device {device_name} critical power draw: {power_draw:.2f} W.
Aborting operation."
```

```
            )

        print(f"Device {device_name}: Power draw {power_draw:.2f} W.")

    except (ValueError, ZeroDivisionError) as e:
        logging.critical(f"Device {device_name}: Calculation error: {e}.
Operation aborted.")
    except TelemetryError as e:
        logging.error(f"Device {device_name}: Operational Error: {e}. System
requires attention.")
    except Exception as e:
        logging.exception(f"Device {device_name}: An unhandled exception
occurred.")


print("--- Logging vs. Exception Throwing ---")

# Test Cases
simulate_device_operation("Heater_Unit", 20.0, 1.5)      # ✅ Normal
simulate_device_operation("Cooling_Fan", -5.0, 1.0)      # ❌ ValueError
simulate_device_operation("Pump_System", 20.0, 0.0)      # ❌ ZeroDivisionError
simulate_device_operation("Turbine_Regulator", 150.0, 4.0)# ❌ TelemetryError
```

Summary: Logging vs. Raising Exceptions

Feature	Logging	Raising Exceptions
Purpose	Record events, monitor status, debug, audit.	Signal critical errors, interrupt abnormal flow.
Flow Control	Program continues execution.	Program flow is immediately interrupted; caller must handle.
Severity	Informational, warnings, non-fatal errors.	Fatal errors, unrecoverable states, invalid conditions.
Who Handles	Logging system (configured by developer).	Calling code (via `try...except`).

Feedback	Passive record in log files/streams.	Active, immediate notification of failure.
Use Case	Monitoring system health, debugging production issues, auditing.	Input validation failures, resource unavailability, fundamental logic errors.

6. Key Takeaways

This session has provided you with essential tools and practices for building robust and resilient Python applications, critical for real-world engineering and scientific systems.

- **Exception Handling Fundamentals**: Mastered the core `try`, `except`, `else`, and `finally` blocks for gracefully managing runtime errors and ensuring cleanup operations.
- **Built-in Exceptions**: Gained familiarity with common Python built-in exception types and understood when they are typically raised.
- **Raising Custom Exceptions**: Learned how to define and raise your own application-specific exception classes, making error handling more explicit, readable, and structured.
- **Assertion-Based Debugging**: Understood the role of `assert` statements for internal sanity checks and identifying programming bugs during development, differentiating their purpose from general exception handling.
- **Logging vs. Exception Throwing**: Gained clarity on when to use the `logging` module for recording non-critical events and warnings versus when to `raise` an exception for critical, flow-interrupting errors that require explicit handling.
- **Robust Code Design**: Developed a mindset for anticipating potential failures, validating inputs, and implementing appropriate error management strategies to make your applications more stable and reliable.

By applying these principles, you can write Python code that is not only functional but also resilient, providing better diagnostics and a more stable operational experience for complex engineering systems.

ISRO URSC – Python Training | Analog Data | Rajath Kumar

📩 **For queries:** rajath@analogdata.ai| 📱 (+91) 96633 53992 | 🌐 https://analogdata.ai