

# Day 1 - Session 4:

## Labs and Application Logic

This notebook is dedicated to hands-on application of the Python concepts learned in Day 1. We will outline practical lab exercises that demonstrate the use of conditional logic, data manipulation, and dictionary lookups in engineering and scientific contexts. The goal is to solidify understanding through problem-solving, with both the conceptual theory and runnable code provided.

### 1. Introduction to Lab Exercises

Theory is best understood when applied. This session focuses on bridging the gap between theoretical Python concepts and their practical utility in real-world engineering and scientific problems. We will present three key lab exercises designed to reinforce your understanding of:

- Conditional logic (if-elif-else)
- Basic arithmetic and data types
- Lists, dictionaries, and iteration (for loops)
- Problem-solving approach and modular thinking

For each lab, we will provide a problem statement, a conceptual approach, and the Python code implementation.

### 2. Lab Exercise 1: Resistance Calculation with Conditional Logic

#### Problem Statement:

Design a Python script that calculates the equivalent resistance of a circuit. The user should be able to specify whether resistors are connected in series or parallel. The script must handle various scenarios, including invalid input for connection type and ensuring that resistance values are positive.

#### Key Concepts to Apply:

- Conditional Statements: if-elif-else to choose between series and parallel calculations.
- Loops: for loop to iterate through multiple resistance values.
- Data Types: float for resistance values, str for connection type.
- Input/Output: input() for user interaction, print() for results and error messages.

- **Error Handling (Basic):** Check for positive resistance values and valid connection type.

#### Formulas:

- **Series Resistance (Req):** Sum of individual resistances.

$$Req=R1+R2+...+Rn$$

- **Parallel Resistance (Req):** Reciprocal of the sum of reciprocals of individual resistances.

$$1/Req=1/R1+1/R2+...+1/Rn$$

Or, for two resistors:  $Req=(R1*R2)/(R1+R2)$  (*more complex for  $n>2$* )

#### Conceptual Approach:

1. **Get Connection Type:** Prompt the user to enter "series" or "parallel".
2. **Validate Connection Type:** Use an **if-else** or **if-elif-else** to check if the input is valid. If not, print an error and exit or re-prompt.
3. **Get Resistance Values:** Ask the user how many resistors they want to enter. Then, use a **for** loop to prompt for each resistance value.
4. **Validate Resistance Values:** Inside the loop, use an **if** statement to ensure each resistance value is positive. If not, print an error and skip or re-prompt for that value.
5. **Calculate Equivalent Resistance:**
  - If "series", sum the resistances.
  - If "parallel", calculate the sum of reciprocals, then take the reciprocal of the total.
6. **Display Result:** Print the calculated equivalent resistance.

Table of Input/Output Examples:

Connection Type	Individual Resistances (Ohms)	Expected Output (Ohms)	Notes
series ▾	10, 20, 30	60.0	Simple sum.
parallel ▾	10, 20	6.67	$1/(1/10+1/20)$
parallel ▾	5, 10, 15	2.73	$1/(1/5+1/10+1/15)$
series ▾	5, -20 (invalid)	Error / Re-prompt	Must handle negative input.
invalid_type ▾	N/A	Error message	Must handle invalid connection type.

## 2.1 Code Implementation: Resistance Calculation

Python

```
def calculate_equivalent_resistance():
    """
    Calculates the equivalent resistance of resistors in series or parallel.
    Handles user input for connection type and validates resistance values.
    """
    print("--- Lab Exercise 1: Resistance Calculation ---")

    while True:
        connection_type = input("Enter connection type (series/parallel): ").strip().lower()
        if connection_type in ["series", "parallel"]:
            break
        else:
            print("Invalid connection type. Please enter 'series' or 'parallel'.")

    resistances = []
    while True:
        num_resistors_str = input("Enter the number of resistors: ")
        try:
            num_resistors = int(num_resistors_str)
            if num_resistors <= 0:
                print("Number of resistors must be positive.")
                continue
```

```

        break
    except ValueError:
        print("Invalid input. Please enter a whole number for the count of
resistors.")

    for i in range(num_resistors):
        while True:
            resistor_value_str = input(f"Enter resistance value for resistor
{i+1} (Ohms): ")
            try:
                resistor_value = float(resistor_value_str)
                if resistor_value <= 0:
                    print("Resistance value must be positive. Please
re-enter.")
                else:
                    resistances.append(resistor_value)
                    break
            except ValueError:
                print("Invalid input. Please enter a numerical value for
resistance.")

    equivalent_resistance = 0.0

    if connection_type == "series":
        equivalent_resistance = sum(resistances)
        print(f"\nResistors in series: {resistances}")
        print(f"Equivalent Resistance (Series): {equivalent_resistance:.2f}
Ohms")
    elif connection_type == "parallel":
        # Handle the case where there are no resistors to avoid division by
zero
        if not resistances:
            print("No resistors provided for parallel calculation.")
            return

        sum_of_reciprocals = 0.0
        for r in resistances:
            # Ensure no division by zero for individual resistors in parallel
            if r == 0:
                print("Warning: Encountered a 0 Ohm resistor in parallel.
Equivalent resistance is 0.")
                equivalent_resistance = 0.0
                break # Short-circuit if one resistor is 0 in parallel
            sum_of_reciprocals += (1 / r)
        else: # This else block executes if the for loop completes without a
break
            if sum_of_reciprocals == 0:

```

```

        # This case implies all resistances were infinite or invalid,
        which should have been caught
        # by validation. But as a safeguard:
        print("Error: Sum of reciprocals is zero. Cannot calculate
        parallel resistance.")
    else:
        equivalent_resistance = 1 / sum_of_reciprocals
        print(f"\nResistors in parallel: {resistances}")
        print(f"Equivalent Resistance (Parallel):
        {equivalent_resistance:.2f} Ohms")

# Uncomment the line below to run Lab Exercise 1
# calculate_equivalent_resistance()

```

### 3. Lab Exercise 2:

## Classification Based on Sensor Ranges

#### Problem Statement:

You are receiving data from a sensor that measures environmental conditions (e.g., radiation levels, air quality index, vibration amplitude). Based on the sensor's reading, classify the condition into predefined categories (e.g., "Safe", "Warning", "Critical"). The system should also provide an alert if the reading is outside any expected range (e.g., due to sensor malfunction).

#### Key Concepts to Apply:

- Conditional Statements: Extensive use of if-elif-else for multi-level classification.
- Logical Operators: and, or for defining ranges.
- Data Types: float for sensor readings, str for classification results.
- Variables: Define clear thresholds as constants.

#### Classification Ranges (Example for a generic sensor reading):

- Invalid/Error: Reading  $<0$  or Reading  $>1000$  (indicating sensor malfunction)
- Safe:  $0 \leq \text{Reading} < 100$
- Warning:  $100 \leq \text{Reading} < 500$
- Critical:  $500 \leq \text{Reading} < 1000$

### Conceptual Approach:

1. **Define Thresholds:** Set constants for the upper and lower bounds of each category.
2. **Get Sensor Reading:** Simulate a sensor reading (e.g., by asking the user for input or picking a value from a predefined list).
3. **Apply Classification Logic:** Use a series of if-elif-else statements to check the reading against the defined ranges in a logical order (e.g., start with the most extreme condition like "Invalid", then "Critical", "Warning", "Safe").
4. **Display Classification:** Print the category the reading falls into and any associated alert messages.
5. **Loop for Multiple Readings:** Consider putting this logic inside a for loop to process a batch of readings or a while loop to continuously monitor.

Table of Sensor Reading Examples and Expected Classifications:

Sensor Reading	Expected Classification	Notes
-5.0	Invalid/Error ▾	Outside expected operating range.
50.0	Safe ▾	Within the safe operating zone.
120.0	Warning ▾	Needs attention, but not critical.
650.0	Critical ▾	Immediate action required.
1000.0	Invalid/Error ▾	Upper bound for valid readings.
99.9	Safe ▾	Edge case near threshold.
100.0	Warning ▾	Edge case exactly on threshold.

### 3.1 Code Implementation: Sensor Reading Classification

Python

```
def classify_sensor_reading():
    """
    Classifies a sensor reading into 'Safe', 'Warning', 'Critical', or
    'Invalid/Error'
    based on predefined ranges.
    """

    print("\n--- Lab Exercise 2: Sensor Reading Classification ---")

    # Define thresholds as constants for easy modification
    MIN_VALID_READING = 0.0
    MAX_VALID_READING = 1000.0 # This reading itself is considered
    invalid/error, bounds are exclusive
    SAFE_THRESHOLD_HIGH = 100.0 # < 100 is Safe
    WARNING_THRESHOLD_HIGH = 500.0 # < 500 is Warning (100 to 499.99)
    # CRITICAL_THRESHOLD_HIGH implicitly starts from 500 up to <1000

    while True:
        reading_str = input("Enter sensor reading (numeric, or 'quit' to exit):
        ").strip().lower()
        if reading_str == 'quit':
            print("Exiting sensor classification.")
            break

        try:
            reading = float(reading_str)

            classification = ""
            alert_message = ""

            # Check for invalid/error range first (most extreme conditions)
            if reading < MIN_VALID_READING or reading >= MAX_VALID_READING:
                classification = "Invalid/Error"
                alert_message = "Reading outside expected operating range!
                Sensor malfunction suspected."
            elif reading >= WARNING_THRESHOLD_HIGH: # Covers 500 up to <1000
                classification = "Critical"
                alert_message = "Immediate action required!"
            elif reading >= SAFE_THRESHOLD_HIGH: # Covers 100 up to <500
                classification = "Warning"
                alert_message = "Needs attention, but not critical."
            else: # Covers 0 up to <100
                classification = "Safe"
```

```
        alert_message = "Within the safe operating zone."

    print(f"Sensor Reading: {reading:.2f}")
    print(f"Classification: {classification}")
    print(f"Alert: {alert_message}\n")

except ValueError:
    print("Invalid input. Please enter a numerical value for the
reading or 'quit'.\n")

# Uncomment the line below to run Lab Exercise 2
# classify_sensor_reading()
```

## 4. Lab Exercise 3:

### Lookup Structures Using Dictionaries

#### Problem Statement:

Develop a system that can quickly retrieve information about various equipment units or error codes using lookup tables. This is common in telemetry systems, inventory management, or diagnostic tools.

#### Key Concepts to Apply:

- **Dictionaries:** Store key-value pairs for efficient lookups.
- **String Manipulation:** For parsing input keys or formatting output.
- **Conditional Logic:** Check if a key exists in the dictionary.
- **Loops:** To process multiple queries or build the lookup table.
- **dict.get() method:** For safe access to dictionary values, providing a default if the key is not found.

#### Scenario Examples:

1. **Equipment ID to Location Mapping:** Map unique equipment IDs to their physical locations.
2. **Error Code to Description Mapping:** Map numerical error codes to detailed human-readable descriptions and recommended actions.



### Conceptual Approach (Error Code Mapping):

1. **Create a Dictionary:** Populate a Python dictionary where keys are error codes (e.g., int or str) and values are their corresponding descriptions and recommended actions (e.g., str or a nested dict).
2. **Get User Input:** Prompt the user to enter an error code they want to look up.
3. **Perform Lookup:** Use the dictionary to retrieve the description.
4. **Handle Missing Keys:** Use `if key in dict` or `dict.get()` to gracefully handle cases where the entered error code does not exist in your dictionary, providing a "not found" message instead of an error.
5. **Loop for Continuous Lookup:** Allow the user to look up multiple codes until they decide to quit.

Table of Error Code Lookup Examples:

Error Code Input	Dictionary Contents (Example)	Expected Output	Notes
101	{101: "Sensor c... ▾	"Sensor calibra... ▾	Direct lookup. ▾
205	{205: "Power s... ▾	"Power supply l... ▾	Direct lookup. ▾
999	(Not in dictionary) ▾	"Error code not ... ▾	Graceful handli... ▾
0	(Not in dictionary) ▾	"Error code not ... ▾	Another missin... ▾

## 4.1 Code Implementation: Error Code Lookup

Python

```
def error_code_lookup():
    """
    Provides a system to look up error codes and their descriptions from a
    dictionary.
    Allows continuous lookups until the user quits.
    """
    print("\n--- Lab Exercise 3: Error Code Lookup ---")

    # Error code dictionary: Key (int) -> Value (dict with description and
    # action)
    error_data = {
        101: {"description": "Sensor calibration required.", "action": "Run
        diagnostic on affected sensor."},
        102: {"description": "Telemetry link intermittent.", "action": "Check
        antenna alignment and signal strength."},
        205: {"description": "Power supply unit low voltage.", "action":
        "Inspect power input, replace PSU if necessary."},
        310: {"description": "Actuator response delayed.", "action": "Check
        hydraulic fluid levels and actuator motor."},
        404: {"description": "Software module not found.", "action": "Verify
        module installation and path configuration."},
        500: {"description": "Critical system overload.", "action": "Immediate
        system shutdown and power cycle. Analyze logs."}
    }

    print("Available Error Codes (for reference):")
    # Print sorted keys for easy reference during the lab
    print(sorted(error_data.keys()))
    print("Type 'quit' to exit lookup.")

    while True:
        code_input = input("Enter error code to lookup: ").strip().lower()

        if code_input == 'quit':
            print("Exiting error code lookup.")
            break

        try:
            error_code = int(code_input)

            # Use dict.get() for safe lookup, providing a default message if
            not found
```

```

        info = error_data.get(error_code)

        if info:
            print(f"\n--- Details for Error Code {error_code} ---")
            print(f"Description: {info['description']}")
            print(f"Recommended Action: {info['action']}")
            print("-----\n")
        else:
            print(f"Error code '{error_code}' not found in the database.
Please check the code.\n")

    except ValueError:
        print("Invalid input. Please enter a numerical error code or
'quit'.\n")

# Uncomment the line below to run Lab Exercise 3
# error_code_lookup()

```

## 5. Challenge/Extension Ideas for Each Lab

These ideas encourage further exploration and application of Python concepts, pushing the boundaries of the basic lab exercises.

### 5.1 Lab Exercise 1: Resistance Calculation Extensions

- **Error Reporting:** Instead of just printing "Invalid input," modify the function to return a specific error message or raise a custom exception that explains *why* the input was invalid (e.g., "Negative resistance value entered").
- **Arbitrary Resistor Combinations:** Allow the user to specify a mix of series and parallel connections within a single circuit (e.g., "R1 in series with (R2 parallel with R3)"). This would require parsing a more complex input string or using a data structure to represent the circuit.
- **Graphical Representation:** (More advanced) After calculating, try to generate a simple text-based or graphical representation of the circuit using a library like matplotlib (even just a simple diagram with labels).
- **File Input:** Modify the function to read resistance values from a text file (e.g., CSV) instead of prompting the user.

## 5.2 Lab Exercise 2: Classification Extensions

- **Batch Processing from File:** Instead of single-reading input, read a CSV file containing multiple sensor readings and apply the classification to each, outputting results to a new file or a structured report.
- **Time-Series Analysis:** Implement logic to detect trends over *multiple consecutive* readings (e.g., "critical" if 3 consecutive readings are in the "warning" range).
- **Dynamic Thresholds:** Allow thresholds to be configured from an external configuration file (e.g., JSON or YAML) instead of hardcoding them.
- **Historical Data Alerts:** Store a history of the last N readings and alert if the rate of change (e.g., (current - previous) / time) exceeds a certain velocity threshold.

## 5.3 Lab Exercise 3: Lookup Structure Extensions

- **Add/Modify/Delete Entries:** Implement administrative functions that allow an authorized user to add new error codes, modify existing descriptions, or delete entries from the error\_data dictionary.
- **Persistent Storage:** Instead of hardcoding the error\_data dictionary, load it from and save it to a file (e.g., JSON or CSV) so changes persist across program runs.
- **Fuzzy Search:** If an exact error code isn't found, implement a "fuzzy" search that suggests similar codes based on string similarity (e.g., using difflib or fuzzywuzzy libraries, though difflib is standard).
- **Multi-criteria Lookup:** For an equipment database, allow users to search for equipment by location, type, or operational status, rather than just ID.

## 6. Key Takeaways

This session focused on translating theoretical Python knowledge into practical, problem-solving code.

- **Application of Control Flow:** You gained hands-on experience using if-elif-else for complex decision-making and for/while loops for iterative processes.
- **Practical Data Handling:** The exercises reinforced the importance of input validation and using appropriate data structures (lists for sequences, dictionaries for lookups).
- **Modular Programming:** Breaking down complex problems (like resistance calculation or sensor classification) into smaller, manageable functions (like `calculate_equivalent_resistance()`) is a key principle for writing clean and reusable code.
- **Problem-Solving Approach:** You practiced analyzing a problem, breaking it into smaller steps, identifying relevant Python constructs, and implementing a robust solution that includes basic error handling.




Mastering these practical application skills is fundamental for developing reliable and efficient solutions in your engineering and scientific endeavors.

### Instructions for Running the Labs:

To run each exercise, **uncomment the corresponding function call** at the end of its code block (e.g., `# calculate_equivalent_resistance()`) and execute the cell in your Jupyter Notebook. You can uncomment one at a time to test them individually.

---

ISRO URSC – Python Training | AnalogData | Rajath Kumar

 For queries: [rajath@analogdata.ai](mailto:rajath@analogdata.ai) |  [\(+91\) 96633 53992](tel:+919663353992) |  <https://analogdata.ai>

---

This material is part of the ISRO URSC Python Training Program conducted by Analog Data (June 2025). For educational use only. © 2025 AnalogData.

---