

Day 1 - Session 3:

Control Flow in Engineering Context - Deep Dive

This notebook explores the fundamental control flow statements in Python (if, for, while), emphasizing their application in typical engineering scenarios. We will delve into conditional logic, iterative processes, and loop control mechanisms, along with practical examples for sensor data processing and pattern generation.

1. Introduction to Control Flow

Control flow statements are the backbone of any programming language, enabling your code to make decisions, execute blocks of code repeatedly, and respond dynamically to varying conditions. In engineering and scientific applications, precise control over execution paths is crucial for handling instrument data, managing processes, and implementing algorithms.

Python's control flow constructs are designed for readability and expressiveness, facilitating the implementation of complex logic.

2. Conditional Statements: **if**, **if-else**, **if-elif-else**

Conditional statements allow your program to execute different blocks of code based on whether certain conditions are met. This is essential for decision-making processes in engineering, such as alarm systems, state machines, or data validation.

2.1 The **if** Statement

The simplest conditional statement. Executes a block of code only if the condition is True.

Python

```
# Scenario: Check if a sensor reading exceeds a critical threshold
temperature_c = 35.5
CRITICAL_TEMP_THRESHOLD = 30.0

if temperature_c > CRITICAL_TEMP_THRESHOLD:
    print(f"WARNING: Temperature ({temperature_c}°C) exceeds critical threshold of {CRITICAL_TEMP_THRESHOLD}°C!")
    print("Initiating cooling sequence.")
```

```

pressure_kpa = 95.0
MIN_PRESSURE_THRESHOLD = 100.0

if pressure_kpa < MIN_PRESSURE_THRESHOLD:
    print(f"ALERT: Pressure ({pressure_kpa} kPa) is below minimum safe level of {MIN_PRESSURE_THRESHOLD} kPa!")

```

2.2 The `if-else` Statement

Executes one block of code if the condition is True, and a different block if it is False.

```

Python
# Scenario: Classify system status based on a single condition
battery_voltage = 23.8 # Volts
NOMINAL_VOLTAGE_MIN = 24.0

if battery_voltage >= NOMINAL_VOLTAGE_MIN:
    print(f"System Status: Nominal (Battery Voltage: {battery_voltage}V)")
else:
    print(f"System Status: Low Battery (Battery Voltage: {battery_voltage}V). Consider charging.")

# Another example: Data validity check
sensor_value = None # Could be a valid number or None if data acquisition failed

if sensor_value is not None:
    print(f"Sensor data received: {sensor_value}")
else:
    print("Error: No sensor data received. Check connection.")

```

2.3 The `if-elif-else` Statement

Allows for multiple conditions to be checked sequentially. The first True condition's block is executed, and subsequent `elif` (else if) and `else` blocks are skipped.

Python

```
# Scenario: Control valve based on flow rate measurements
flow_rate_LPM = 12.5 # Liters Per Minute
LOW_FLOW_THRESHOLD = 5.0
HIGH_FLOW_THRESHOLD = 15.0

if flow_rate_LPM < LOW_FLOW_THRESHOLD:
    print(f"Flow rate ({flow_rate_LPM} LPM) is too low. Opening valve wider.")
elif flow_rate_LPM > HIGH_FLOW_THRESHOLD:
    print(f"Flow rate ({flow_rate_LPM} LPM) is too high. Closing valve slightly.")
else:
    print(f"Flow rate ({flow_rate_LPM} LPM) is optimal. Maintaining valve position.")

print("\n--- Another example: Categorizing signal strength ---")
signal_strength_db = -75 # dBm

# Using logical operators (and, or, not)
if signal_strength_db >= -60:
    print("Signal Strength: Excellent")
elif -80 <= signal_strength_db < -60: # equivalent to signal_strength_db >= -80
    and signal_strength_db < -60
    print("Signal Strength: Good")
elif -90 <= signal_strength_db < -80:
    print("Signal Strength: Fair")
else:
    print("Signal Strength: Poor or No Signal")
```

Summary of Conditional Statements

Statement	Description	When to Use
`if`	Executes code if a single condition is True.	Simple condition checks (e.g., alarm triggers, data presence).
`if-else`	Executes one block if True, another if False.	Binary decision points (e.g., pass/fail, active/inactive).
`if-elif-else`	Checks multiple conditions sequentially.	Multi-way decision trees, categorizing data into several bins.

2.4 Ternary Conditional Operator & Nested Conditions

2.4.1 Ternary Conditional Operator (Tertiary Conditions)

This is a concise way to write if-else statements in a single line, often used for assigning values conditionally.

Python

```
# Syntax: value_if_true if condition else value_if_false

# Scenario: Determine component status based on test result
test_result = 0.85 # Pass if >= 0.9, else Fail
required_accuracy = 0.9

component_status = "Pass" if test_result >= required_accuracy else "Fail"
print(f"Component Test Status: {component_status}")

# Example with numerical output
current_load_percent = 78
max_allowed_temp_c = 120 if current_load_percent < 80 else 100
print(f"Maximum Allowed Temperature for {current_load_percent}% load:
{max_allowed_temp_c}°C")

# Nested ternary (can become hard to read, use sparingly)
# Scenario: Categorize pressure: High, Low, or Optimal, also considering a
'critical' threshold
pressure_reading = 105.0 # kPa
OPTIMAL_LOW = 95.0
OPTIMAL_HIGH = 105.0
CRITICAL_HIGH = 110.0

pressure_category = (
    "Critical High" if pressure_reading >= CRITICAL_HIGH else
    "High" if pressure_reading > OPTIMAL_HIGH else
    "Low" if pressure_reading < OPTIMAL_LOW else
    "Optimal"
)
print(f"Pressure category for {pressure_reading} kPa: {pressure_category}")
```

2.4.2 Nested `if` Statements

When you have conditions that depend on other conditions, you can nest `if` statements. While powerful, excessive nesting can reduce readability.

Python

```
# Scenario: Control a robotic arm based on multiple sensor inputs
arm_position_ok = True
safety_lock_engaged = False
target_reached = False
obstacle_detected = True

print("\n--- Robotic Arm Control Logic ---")
if arm_position_ok:
    print("Arm position is within limits.")
    if not safety_lock_engaged:
        print("Safety lock is disengaged.")
        if not obstacle_detected:
            print("No obstacles detected.")
            if not target_reached:
                print("Initiating movement to target.")
            else:
                print("Target already reached. Arm is idle.")
        else:
            print("WARNING: Obstacle detected! Halting arm movement.")
    else:
        print("Safety lock is engaged. Cannot move arm.")
else:
    print("ERROR: Arm position is out of limits! Emergency stop initiated.")

# Example with `elif` in nested structures for clarity
print("\n--- Complex Valve Control based on Pressure and Flow ---")
current_pressure = 102.5 # kPa
current_flow = 8.0 # LPM

if current_pressure > 105.0:
    print("Pressure is too high.")
    if current_flow < 5.0:
        print(" -> Low flow despite high pressure. Possible blockage detected. Shutting valve.")
    else:
        print(" -> High flow with high pressure. Opening relief valve.")
elif current_pressure < 95.0:
    print("Pressure is too low.")
    if current_flow > 10.0:
        print(" -> High flow despite low pressure. Possible leak detected.")
```

```

Closing main valve.")
    else:
        print(" -> Low flow with low pressure. Increasing pump speed.")
else:
    print("Pressure is within optimal range.")
    if 7.0 <= current_flow <= 9.0:
        print(" -> Flow is also optimal. Maintaining system stability.")
    else:
        print(" -> Flow is outside optimal range. Adjusting flow regulator.")

```

3. Looping Statements: **for**, **while**

Loops are used to execute a block of code repeatedly. This is fundamental for processing collections of data, iterating through sensor streams, or performing calculations until a certain condition is met.

3.1 The for Loop

Iterates over a sequence (like a list, tuple, string, or range) or any other iterable object.

```

Python
# Scenario: Process a batch of sensor readings
daily_temperatures_c = [24.1, 25.3, 23.8, 26.0, 24.5]

print("Processing daily temperatures:")
for temp in daily_temperatures_c:
    print(f"Temperature reading: {temp}°C")
    # Simulate a calculation
    temp_f = (temp * 9/5) + 32
    print(f" -> Equivalent in Fahrenheit: {temp_f:.2f}°F")

print("\n--- Iterating with index using `enumerate` ---")
# When you need both the item and its index (e.g., for logging line numbers)
log_messages = [
    "Sensor data received.",
    "Processing anomaly.",
    "Calibration required."
]

```

```

for i, message in enumerate(log_messages):
    print(f"Log {i+1}: {message}")

print("\n--- Iterating over a range with a step ---")
# Scenario: Accessing every other data point for quick checks
sensor_readings_long = [
    10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8, 10.9, 11.0,
    11.1, 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.8, 11.9, 12.0
]
print("Checking every 3rd sensor reading (starting from 0th):")
for i in range(0, len(sensor_readings_long), 3):
    print(f"  Reading at index {i}: {sensor_readings_long[i]}")

print("\n--- Nested `for` loops: Processing a 2D grid/matrix ---")
# Scenario: Analyze data from a grid of sensors (e.g., heat map, pressure
array)
sensor_grid_data = [
    [100, 105, 110, 108],
    [98, 102, 107, 101],
    [103, 99, 100, 104]
]
CRITICAL_GRID_TEMP = 105

print("Analyzing sensor grid for hot spots:")
for row_idx, row in enumerate(sensor_grid_data):
    for col_idx, temp in enumerate(row):
        if temp > CRITICAL_GRID_TEMP:
            print(f"  HOT SPOT detected at Row {row_idx}, Col {col_idx}:
{temp}°C")
        else:
            print(f"  Normal at Row {row_idx}, Col {col_idx}: {temp}°C")

```

Use Cases for for Loops:

- Processing each item in a list of sensor readings.
- Iterating through characters in a message.
- Performing a fixed number of iterations for a simulation.
- Traversing elements in a matrix (nested loops).

3.2 The `while` Loop

Executes a block of code as long as a specified condition is True. It's crucial for indefinite loops where the number of iterations isn't known beforehand (e.g., waiting for a sensor reading, polling a device).

Python

```
# Scenario: Continuously monitor a system until a fault occurs
system_operational = True
sensor_value = 0
MAX_SAFE_VALUE = 100

print("Monitoring system...")
while system_operational:
    # Simulate reading sensor data
    sensor_value += 10 # Increment sensor value over time

    if sensor_value > MAX_SAFE_VALUE:
        system_operational = False # Condition becomes False, loop terminates
        print(f"FAULT DETECTED: Sensor value ({sensor_value}) exceeded max safe
value ({MAX_SAFE_VALUE}). Shutting down.")
    else:
        print(f"Current sensor value: {sensor_value}. System operational.")

    # In a real system, you'd have a time delay here to prevent infinite
loop/resource exhaustion
    # For demonstration, we just increment and let it run quickly
    if sensor_value > 200: # Prevent excessive output in notebook
        break # Emergency break for demonstration purposes

print("System monitoring stopped.")

print("\n--- Another example: Retrying a connection with increasing delay ---")
connection_attempts = 0
MAX_ATTEMPTS = 5
connected = False
delay_seconds = 1 # Initial delay

while not connected and connection_attempts < MAX_ATTEMPTS:
    connection_attempts += 1
    print(f"Attempting to connect (Attempt {connection_attempts})...")
    # Simulate connection logic (e.g., trying to ping an IP)
    # In real code, you'd use time.sleep(delay_seconds)
    # import time
    # time.sleep(delay_seconds)
```



```

    if connection_attempts == 3: # Simulate successful connection on 3rd
attempt
        connected = True
        print("Connection successful!")
    else:
        print(f"Connection failed. Waiting {delay_seconds} seconds before
retrying.")
        delay_seconds *= 2 # Exponential backoff
        if delay_seconds > 8: # Cap the max delay
            delay_seconds = 8

if not connected:
    print(f"Failed to connect after {MAX_ATTEMPTS} attempts.")

```

Use Cases for while Loops:

- Continuous data acquisition until a stop signal.
- Implementing retry mechanisms for network connections or sensor reads.
- Iterating until a specific calculation converges (e.g., numerical solvers).
- Implementing game loops.

4. Loop Control Statements: **break**, **continue**, **else**

These statements give you finer control over loop execution.

4.1 The **break** Statement

Terminates the loop entirely and immediately transfers control to the statement following the loop.

```

Python
# Scenario: Find the first anomaly in a data stream
data_stream = [10.1, 10.2, 10.3, 15.5, 10.4, 10.5] # 15.5 is an anomaly
ANOMALY_THRESHOLD = 15.0
anomaly_found = False

print("Scanning data stream for anomalies...")

```

```

for i, value in enumerate(data_stream):
    if value > ANOMALY_THRESHOLD:
        print(f"Anomaly detected at index {i}: Value = {value}")
        anomaly_found = True
        break # Exit the loop immediately after finding the first anomaly
    print(f"Processing data point {i}: {value}")

if not anomaly_found:
    print("No anomalies detected in the stream.")

print("\n--- Using `break` in a `while` loop for user input ---")
while True: # Infinite loop until break
    user_input = input("Enter sensor command (or 'exit' to quit):")
    user_input = user_input.strip().lower()
    if user_input == 'exit':
        print("Exiting command interface.")
        break
    print(f"Executing command: '{user_input}'")
    # Add actual command processing logic here

```

4.2 The `continue` Statement

Skips the rest of the current iteration of the loop and moves to the next iteration.

Python

```

# Scenario: Process valid sensor readings, skip corrupted ones
raw_sensor_values = [120, 'corrupted', 155, 98, 'invalid_format', 210]
processed_readings = []
print("Processing raw sensor values, skipping invalid entries:")
for value in raw_sensor_values:
    if not isinstance(value, (int, float)): # Check if value is not a number
        print(f"Skipping invalid data: {value}")
        continue # Skip to the next iteration

    # If we reach here, the value is valid
    processed_readings.append(value)
    print(f"Processed valid reading: {value}")

print(f"Final processed readings: {processed_readings}")

```

4.3 The `else` Clause with Loops (`for-else`, `while-else`)

Python loops have an optional `else` block.

- **`for-else`**: The `else` block executes only if the loop completes without encountering a `break` statement.
- **`while-else`**: The `else` block executes only if the `while` loop condition becomes `False` (i.e., it terminates normally) and not by a `break` statement.

This is useful for "search" operations where you want to know if an item was found or if the entire sequence was traversed without a match.

```
Python
print("--- `for-else` Example: Searching for a specific component ---")
component_list = ["Resistor_A", "Capacitor_B", "Diode_C", "Inductor_D"]
target_component = "Diode_C"
# target_component = "Transistor_E" # Uncomment to see else block execute

for component in component_list:
    if component == target_component:
        print(f"Component '{target_component}' found!")
        break # If found, break the loop
else:
    # This 'else' block runs only if the loop completes naturally (no break)
    print(f"Component '{target_component}' not found in the list.")

print("\n--- `while-else` Example: Successful connection without interruption
---")
attempts = 0
max_attempts = 3
device_ready = False

while attempts < max_attempts:
    attempts += 1
    print(f"Checking device status... (Attempt {attempts})")
    # Simulate a successful check on the 3rd attempt
    if attempts == 3:
        device_ready = True
        break # If ready, break the loop
else:
    # This 'else' block runs if attempts >= max_attempts (loop finishes
    normally)
    if not device_ready:
```

```

        print("Device not ready after maximum attempts.")

if device_ready:
    print("Device is ready for operation.")

```

4.4 Tricky Examples: `break`/`continue` in Nested Loops

When dealing with nested loops, `break` and `continue` only affect the innermost loop they are in.

```

Python
# Scenario: Find the first invalid data point in a 2D sensor array and stop
processing that row
sensor_data_grid = [
    [10.1, 10.2, 10.3, 10.4],
    [20.1, 20.2, 'INVALID', 20.4], # Invalid data
    [30.1, 30.2, 30.3, 30.4]
]

print("--- Searching for first invalid data point per row (using `break`) ---")
for row_idx, row in enumerate(sensor_data_grid):
    print(f"Processing Row {row_idx}:")
    for col_idx, value in enumerate(row):
        if not isinstance(value, (int, float)):
            print(f"  Invalid data '{value}' found at ({row_idx}, {col_idx}).
Stopping processing for this row.")
            break # Breaks only the inner loop (over columns)
            print(f"  Valid data at ({row_idx}, ({col_idx}): {value}")
        else: # This else belongs to the inner for loop
            print(f"  Row {row_idx} processed completely (no invalid data).")

print("\n--- Filtering invalid data in nested loops (using `continue`) ---")
# Scenario: Calculate average of valid readings in each row, skipping invalid
ones
filtered_averages = []
for row_idx, row in enumerate(sensor_data_grid):
    valid_readings_in_row = []
    for col_idx, value in enumerate(row):
        if not isinstance(value, (int, float)):
            print(f"  Skipping invalid data '{value}' at ({row_idx},
{col_idx}).")
            continue # Skips to the next column in the current row

```

```

        valid_readings_in_row.append(value)

    if valid_readings_in_row:
        avg = sum(valid_readings_in_row) / len(valid_readings_in_row)
        filtered_averages.append(avg)
        print(f"  Average of valid readings in Row {row_idx}: {avg:.2f}")
    else:
        print(f"  No valid readings found in Row {row_idx}.")

print(f"Averages of valid readings per row: {filtered_averages}")

```

Summary of Loop Control Statements

Statement	Description	Effect	When to Use
<code>`break`</code>	Terminates the loop immediately.	Jumps out of the current loop.	Found what you're looking for, or an error condition occurs.
<code>`continue`</code>	Skips the rest of the current iteration.	Proceeds to the next iteration of the current loop.	Skip invalid data, process specific conditions within a loop.
<code>`else`</code> (with <code>`for`/`while`</code>)	Executes if the loop completes normally (no break).	Provides a way to check if a "search" was successful.	Confirming all items processed, or if a target was not found after full traversal.

5. Pattern Generation and Conditional Logic for Sensors

In engineering, sensor data often arrives in streams, and you need to apply conditional logic to identify patterns, generate alerts, or trigger actions.

5.1 Simple Threshold-Based Pattern Detection

Python

```
# Scenario: Detect if a temperature goes above a high threshold or below a low threshold
current_temp_readings = [22.1, 23.5, 25.0, 28.2, 31.0, 29.5, 24.0, 18.0, 19.2]
TEMP_HIGH_THRESHOLD = 30.0
TEMP_LOW_THRESHOLD = 20.0

print("Monitoring temperature readings for abnormal patterns:")
for i, temp in enumerate(current_temp_readings):
    if temp > TEMP_HIGH_THRESHOLD:
        print(f"Reading {i}: {temp}°C - HIGH TEMPERATURE ALERT!")
    elif temp < TEMP_LOW_THRESHOLD:
        print(f"Reading {i}: {temp}°C - LOW TEMPERATURE WARNING!")
    else:
        print(f"Reading {i}: {temp}°C - Normal.")
```

5.2 Detecting Sequences and Trends

More complex patterns involve looking at a sequence of readings or a trend.

Python

```
# Scenario: Detect an "increasing trend" in pressure for 3 consecutive readings
pressure_history = [100.0, 101.5, 102.0, 103.5, 102.8, 104.0, 105.0, 106.0, 105.5]
consecutive_increase_count = 0
REQUIRED_INCREASE_COUNT = 3

print("\nDetecting increasing pressure trends:")
for i in range(1, len(pressure_history)):
    current_pressure = pressure_history[i]
    previous_pressure = pressure_history[i-1]
```

```

if current_pressure > previous_pressure:
    consecutive_increase_count += 1
    print(f"Index {i}: Pressure increased from {previous_pressure} to
{current_pressure}. Consecutive increases: {consecutive_increase_count}")
    if consecutive_increase_count >= REQUIRED_INCREASE_COUNT:
        print(f"*** ALERT: Sustained increasing pressure trend detected for
{REQUIRED_INCREASE_COUNT} readings! ***")
        # Reset count after alert to look for new trends or prevent
repeated alerts
        consecutive_increase_count = 0
    else:
        # Reset count if pressure does not increase
        if consecutive_increase_count > 0:
            print(f"Index {i}: Pressure change not increasing. Resetting
consecutive count ({consecutive_increase_count}).")
            consecutive_increase_count = 0

```

5.3 Implementing a Simple State Machine with Control Flow

Control flow can manage the states of a system based on sensor inputs or internal logic.

Python

```

# Scenario: Simple device state machine (e.g., a sensor that can be OFF,
STARTING, RUNNING, ERROR)
# States: 0=OFF, 1=STARTING, 2=RUNNING, 3=ERROR

current_state = 0 # Initial state: OFF
command_sequence = [
    "START",          # Command to start
    "INIT_SUCCESS",   # Internal signal for successful initialization
    "DATA_FLOW_OK",   # Internal signal for data flow
    "ANOMALY_DETECTED", # Simulate an error
    "RESET",          # Command to reset
    "START"
]

print("Device State Machine Log:")
for command in command_sequence:
    print(f"\nReceived Command: '{command}' (Current State: {current_state})")

    if current_state == 0: # OFF state

```

```




if command == "START":
    print("Transitioning from OFF to STARTING.")
    current_state = 1
else:
    print("Invalid command for OFF state.")
elif current_state == 1: # STARTING state
    if command == "INIT_SUCCESS":
        print("Transitioning from STARTING to RUNNING.")
        current_state = 2
    elif command == "RESET":
        print("Transitioning from STARTING to OFF.")
        current_state = 0
    else:
        print("Waiting for initialization signal.")
elif current_state == 2: # RUNNING state
    if command == "ANOMALY_DETECTED":
        print("Transitioning from RUNNING to ERROR.")
        current_state = 3
    elif command == "RESET":
        print("Transitioning from RUNNING to OFF.")
        current_state = 0
    else:
        print("Device is running normally.")
elif current_state == 3: # ERROR state
    if command == "RESET":
        print("Transitioning from ERROR to OFF.")
        current_state = 0
    else:
        print("Device is in ERROR state. Manual intervention required (or
'RESET').")
else:
    print("Unknown state encountered. Resetting to OFF.")
    current_state = 0

print(f"\nFinal Device State: {current_state}")

```

This notebook illustrates the power and flexibility of Python's control flow statements for implementing decision-making and iterative processes, which are fundamental to scientific and engineering programming. Mastering these constructs allows for the creation of robust and responsive systems, from simple data filters to complex state-machines and real-time monitoring applications.

ISRO URSC – Python Training | AnalogData | Rajath Kumar

 For queries: 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.
