

Day 2 Session 4:

Hands-On Labs - Deep Dive

This notebook is dedicated to hands-on application of the advanced Python concepts covered in Day 2. We will implement real-world engineering examples using decorators, closures, and modular programming techniques to solidify understanding and demonstrate their practical utility.

1. Introduction to Hands-On Labs

The advanced concepts of functions, closures, and decorators, along with modular code organization, are best understood through practical application. This session provides three hands-on lab exercises designed to consolidate your learning from Day 2 and equip you with practical skills for building robust and efficient Python applications in scientific and engineering domains.

Each lab will present a problem, guide you through a conceptual approach, and provide the Python code implementation. Remember to run the code cells and experiment with them to deepen your understanding.

2. Lab Exercise 1:

Decorator to Time Sensor Calculations

Problem Statement:

In scientific and engineering simulations, it's often crucial to monitor the performance of individual calculation steps. Create a Python decorator that measures the execution time of any given function and logs this time to the console. Apply this decorator to a function that simulates a complex sensor data processing task.

Key Concepts to Apply:

- **Decorators:** Writing a higher-order function that wraps another function.
- **@decorator_name syntax:** Applying the decorator.
- ***args and **kwargs:** To allow the decorated function to accept any arguments.
- **functools.wraps:** To preserve the original function's metadata (`__name__`, `__doc__`).
- **time module:** For measuring execution time.

Conceptual Approach:

1. Define a decorator function, `time_function_execution`, that takes a function `func` as an argument.
2. Inside the decorator, define a nested `wrapper` function that accepts `*args` and `**kwargs`.
3. Use `time.perf_counter()` to record the start time before calling the original `func`.
4. Call the `func` with `*args` and `**kwargs` and store its result.
5. Record the end time after `func` completes.
6. Calculate the elapsed time and print a formatted message including the function's name and execution time.
7. Return the result of the original `func` from the `wrapper`.
8. Return the `wrapper` function from the decorator.
9. Apply `@time_function_execution` to a sample function that simulates a sensor calculation (e.g., averaging a large dataset, a matrix operation).

Timing Decorator Behavior

Action	Expected Log Output
Decorator application	No direct output, just wraps function.
Decorated function call	INFO: Function <code>my_func</code> executed in X.XXXs.

2.1 Code Implementation: Timing Decorator

```
Python
import functools
import time
import random

def time_function_execution(func):
    """
    A decorator that measures the execution time of the decorated function.
```

```

    Logs the function name and its execution duration.
    """

@functools.wraps(func) # Preserves original function's metadata
def wrapper(*args, **kwargs):
    start_time = time.perf_counter() # High-resolution timer
    result = func(*args, **kwargs) # Execute the original function
    end_time = time.perf_counter()
    elapsed_time = end_time - start_time
    print(f"INFO: Function '{func.__name__}' executed in {elapsed_time:.4f}
seconds.")
    return result
return wrapper

@time_function_execution
def simulate_complex_sensor_analysis(data_points_count: int,
processing_intensity: float) -> float:
    """
    Simulates a complex sensor data analysis process.
    The execution time depends on data_points_count and processing_intensity.
    """

    print(f" Starting analysis for {data_points_count} data points with
intensity {processing_intensity}...")

    total_sum = 0.0
    for i in range(data_points_count):
        # Simulate some data processing work
        total_sum += random.uniform(0, 1) * processing_intensity
        if i % (data_points_count // 10 or 1) == 0:
            # This print is internal to simulation, not decorator's output
            # print(f" ... processing {i}/{data_points_count}")
            pass # Keep it silent for cleaner output

    time.sleep(data_points_count / 1000000 * processing_intensity) # Simulate
actual work

    print(f" Analysis complete. Total sum simulated: {total_sum:.2f}")
    return total_sum

@time_function_execution
def read_instrument_settings(num_settings: int) -> dict:
    """
    Simulates reading multiple settings from a connected instrument.
    """

    print(f" Reading {num_settings} instrument settings...")
    settings = {}
    for i in range(num_settings):
        settings[f"Setting_{i+1}"] = random.randint(1, 100)

```

```

        time.sleep(0.01) # Simulate I/O delay per setting
    print("  Instrument settings read.")
    return settings

print("--- Lab 1: Decorator to Time Sensor Calculations ---")

# Run the decorated functions
result_1 = simulate_complex_sensor_analysis(100000, 0.5)
print(f"Simulated Analysis Result 1: {result_1:.2f}\n")

result_2 = read_instrument_settings(5)
print(f"Instrument Settings Result 2: {result_2}\n")

result_3 = simulate_complex_sensor_analysis(500000, 1.0) # More data points,
higher intensity
print(f"Simulated Analysis Result 3: {result_3:.2f}\n")

# Expected Output from Decorator:
# INFO: Function 'simulate_complex_sensor_analysis' executed in X.XXX seconds.
# INFO: Function 'read_instrument_settings' executed in Y.YYY seconds.
# INFO: Function 'simulate_complex_sensor_analysis' executed in Z.ZZZ seconds.

```

2.2 Challenge/Extension Ideas for Lab 1

- **Configurable Logger:** Modify the decorator to accept arguments (e.g., `@time_function_execution(log_to_file=True, precision=2)`) to control logging behavior. This will require an extra nested layer in the decorator.
- **Threshold Alert:** Add logic within the decorator to print an "ALERT" if the function execution time exceeds a predefined threshold.
- **Cumulative Timing:** For a class with multiple decorated methods, implement a way for the decorator to keep track of the *total* time spent across all calls to *all* decorated methods in that class. (Hint: might need a class-based decorator or a closure with a shared state).
- **Specific Context Logging:** Include details about the arguments passed to the function in the log output (e.g., `Function 'read_data' called with sensor_id='A01', length=100.`).

3. Lab Exercise 2:

Closure for Adaptive Filter Configuration

Problem Statement:

Design an "adaptive" digital filter configuration function using a closure. The filter should have a base configuration, but certain parameters (e.g., gain) can be "tuned" based on an initial setup value. The closure should allow you to create specialized filter instances that remember their initial tuning parameter, enabling an "adaptive" behavior without modifying the core filter logic directly.

Key Concepts to Apply:

- **Closures:** A nested function remembering variables from its enclosing scope.
- **First-Class Functions:** Returning a function from another function.
- **Default Arguments:** For optional base filter parameters.

Conceptual Approach:

1. Define an outer function, `create_adaptive_filter_configurator`, that takes a `base_gain` as an argument. This `base_gain` will be the closed-over variable.
2. Inside this outer function, define a nested function, `get_filter_params`.
3. `get_filter_params` should take sensor-specific tuning parameters (e.g., `adaptive_factor`) and calculate the final gain (e.g., `final_gain = base_gain * adaptive_factor`). It might also include other static filter parameters.
4. The outer function should return `get_filter_params`.
5. Create several instances of `get_filter_params` by calling `create_adaptive_filter_configurator` with different `base_gain` values.
6. Demonstrate how each specialized filter configurator remembers its `base_gain` and produces different final gains based on the `adaptive_factor`.

Adaptive Filter Configuration

Base Gain (Closure)	Adaptive Factor (Call)	Expected Final Gain
1.0 ▾	0.5 ▾	0.5
1.0 ▾	1.5 ▾	1.5
2.0 ▾	0.5 ▾	1.0
2.0 ▾	1.5 ▾	3.0

3.1 Code Implementation: Closure for Adaptive Filter

Python

```
def create_adaptive_filter_configurator(base_gain: float, filter_type: str =
"LowPass") -> callable:
    """
    Creates and returns a specialized filter configuration function (a
    closure).
    This closure remembers the `base_gain` and `filter_type` from its creation.

    Args:
        base_gain: The initial gain value for the filter, which will be
        adapted.
        filter_type: The type of filter (e.g., "LowPass", "HighPass").

    Returns:
        A callable (function) that takes an `adaptive_factor` and returns
        a dictionary of configured filter parameters.
    """
    if not (0 < base_gain < 100): # Simple validation for base_gain
        raise ValueError("Base gain must be between 0 and 100.")

    def get_filter_params(adaptive_factor: float, cut_off_freq_hz: float =
100.0) -> dict:
        """
        Calculates and returns adaptive filter parameters.
        Closes over `base_gain` and `filter_type`.

        Args:
            adaptive_factor: A multiplier applied to the base_gain for tuning.
            cut_off_freq_hz: The cutoff frequency of the filter in Hz.
```

```

    Returns:
        A dictionary of filter parameters.
    """
    if not (0.1 <= adaptive_factor <= 5.0): # Validation for
adaptive_factor
        print(f"WARNING: Adaptive factor {adaptive_factor} is out of
typical range (0.1-5.0).")

    final_gain = base_gain * adaptive_factor

    # Simulate some logic for cutoff frequency based on type
    if filter_type == "LowPass":
        adjusted_cut_off = min(cut_off_freq_hz, 5000.0) # Cap low-pass
cutoff
    elif filter_type == "HighPass":
        adjusted_cut_off = max(cut_off_freq_hz, 10.0) # Min high-pass
cutoff
    else:
        adjusted_cut_off = cut_off_freq_hz # Default

    return {
        "filter_type": filter_type,
        "final_gain": final_gain,
        "cut_off_frequency_hz": adjusted_cut_off,
        "tuning_parameters": {
            "base_gain_used": base_gain,
            "adaptive_factor_applied": adaptive_factor
        }
    }

    return get_filter_params

print("--- Lab 2: Closure for Adaptive Filter Configuration ---")

# Create specialized filter configurators
# Configurator for a sensor with a base gain of 1.0 (e.g., for low-noise
signals)
config_for_low_noise = create_adaptive_filter_configurator(base_gain=1.0,
filter_type="BandPass")

# Configurator for a sensor needing higher base gain (e.g., for weak signals)
config_for_weak_signal = create_adaptive_filter_configurator(base_gain=2.5,
filter_type="LowPass")

# Use the specialized configurators
print("\n--- Low Noise Sensor Configs ---")

```

```

params_1 = config_for_low_noise(adaptive_factor=0.8, cut_off_freq_hz=150.0)
print(f" Config 1: {params_1}")

params_2 = config_for_low_noise(adaptive_factor=1.2, cut_off_freq_hz=200.0)
print(f" Config 2: {params_2}")

print("\n--- Weak Signal Sensor Configs ---")
params_3 = config_for_weak_signal(adaptive_factor=0.6, cut_off_freq_hz=80.0)
print(f" Config 3: {params_3}")

params_4 = config_for_weak_signal(adaptive_factor=1.5, cut_off_freq_hz=120.0)
print(f" Config 4: {params_4}")

# Demonstrate error handling for base_gain
try:
    invalid_config = create_adaptive_filter_configurator(base_gain=-5.0)
except ValueError as e:
    print(f"\nCaught expected error: {e}")

# Demonstrate accessing closure contents (for introspection)
# Note: Accessing __closure__ is for introspection/debugging, not typical
# production code
if config_for_low_noise.__closure__:
    print(f"\nIntrospection: config_for_low_noise closes over: "
          f"base_gain={config_for_low_noise.__closure__[0].cell_contents}, "
          f"filter_type='{config_for_low_noise.__closure__[1].cell_contents}'")

```

3.2 Challenge/Extension Ideas for Lab 2

- **Complex Adaptive Logic:** Make `get_filter_params` more sophisticated. Instead of a simple multiplication, have it use `adaptive_factor` to look up settings from a nested dictionary or apply a non-linear adaptation curve.
- **Multiple Closed-Over Variables:** Add more parameters to the outer function (e.g., `default_cutoff_freq`, `gain_offset`) that the inner function also closes over.
- **Validation within Closure:** Add more robust input validation within the `get_filter_params` function for `adaptive_factor` and `cut_off_freq_hz`.
- **Stateful Closure:** Modify the closure to maintain an internal state (e.g., a counter for how many times it has been called, or a running average of applied adaptive factors).

4. Lab Exercise 3:

Modular Temperature Logger

Problem Statement:

Build a modular temperature logging system for an embedded device. The system should consist of several independent functions for:

1. **Simulating a temperature reading.**
2. **Validating the reading against a safe range.**
3. **Formatting the reading into a log string.**
4. **Writing the log string to a conceptual log file (simulated by printing to console).**
5. **An orchestrator function that combines these modules to perform a logging cycle.**

Emphasize clear function boundaries, appropriate return values, and type hints for robustness.

Key Concepts to Apply:

- **Writing Modular Functions:** Adhering to the Single Responsibility Principle (SRP).
- **Docstrings and Type Hints:** For clear function interfaces.
- **Conditional Logic (if-else):** For validation.
- **datetime module:** For timestamps.
- **random module:** For simulating readings.
- **while loop:** For continuous logging.

Conceptual Approach:

1. **`simulate_temperature_reading() -> float`:** Returns a random float within a realistic temperature range (e.g., 0-50°C).
2. **`is_reading_valid(temperature: float, min_safe: float, max_safe: float) -> bool`:** Takes a temperature and min/max safe thresholds, returns True if valid, False otherwise.
3. **`format_log_entry(timestamp: str, temperature: float, status: str) -> str`:** Takes time, temp, and status, returns a formatted log string.
4. **`write_to_log(log_entry: str) -> None`:** Takes a log entry string and prints it, simulating writing to file.

5. `run_temperature_logger(duration_seconds: int, interval_seconds: float) -> None:`
The main function. It continuously calls the other functions within a `while` loop for a specified duration, logging readings at defined intervals. It should handle invalid readings by logging an appropriate "Invalid" status.

Modular Logger Flow

Function Call	Input	Output / Action
<code>`simulate_temperature_reading()`</code>	None	<code>`float`</code> (simulated temperature)
<code>`is_reading_valid()`</code>	<code>`float`, `float`, `float`</code>	<code>`bool`</code> (validity)
<code>`format_log_entry()`</code>	<code>`str`, `float`, `str`</code>	<code>`str`</code> (formatted log string)
<code>`write_to_log()`</code>	<code>`str`</code>	Prints log string to console
<code>`run_temperature_logger()`</code>	<code>`int`, `float`</code>	Orchestrates calls, logs continuously for duration.

4.1 Code Implementation: Modular Temperature Logger

```
Python
import datetime
import random
import time
from typing import Tuple

# Define safe operational limits for temperature sensor
MIN_SAFE_TEMP_C = 10.0
MAX_SAFE_TEMP_C = 40.0

def simulate_temperature_reading(min_val: float = 0.0, max_val: float = 50.0)
-> float:
    """
    Simulates reading a temperature from a sensor.
    Returns a random float within the specified range.
    """
```

```

        # Introduce occasional "faulty" readings for testing validation
        if random.random() < 0.05: # 5% chance of an extreme reading
            return random.choice([-100.0, 100.0, 500.0]) # Simulate sensor
malfunction
        return random.uniform(min_val, max_val)

def is_reading_valid(temperature: float, min_safe: float, max_safe: float) ->
Tuple[bool, str]:
    """
    Validates a temperature reading against safe operational limits.
    Returns a tuple: (is_valid: bool, status_message: str).
    """
    if temperature < min_safe:
        return False, "LOW_ALERT"
    elif temperature > max_safe:
        return False, "HIGH_ALERT"
    else:
        return True, "NORMAL"

def format_log_entry(timestamp: str, sensor_id: str, temperature: float,
status: str) -> str:
    """
    Formats a temperature reading into a standardized log string.
    """
    return f"[{timestamp}] SensorID: {sensor_id}, Temp: {temperature:.2f}°C,
Status: {status}"

def write_to_log(log_entry: str) -> None:
    """
    Simulates writing a log entry to a file by printing it to the console.
    In a real system, this would write to a file or a logging service.
    """
    print(log_entry)

def run_temperature_logger(
    sensor_id: str,
    duration_seconds: int,
    interval_seconds: float,
    min_safe_temp: float = MIN_SAFE_TEMP_C,
    max_safe_temp: float = MAX_SAFE_TEMP_C
) -> None:
    """
    Orchestrates the modular temperature logging process.

```

```

    Continuously logs sensor readings for a specified duration.
    """
    print(f"\n--- Starting Temperature Logger for {sensor_id} ---")
    print(f"Logging for {duration_seconds} seconds at {interval_seconds}s
intervals.")
    print(f"Safe Range: {min_safe_temp}°C - {max_safe_temp}°C")

    start_time = time.time()

    while time.time() - start_time < duration_seconds:
        current_timestamp = datetime.datetime.now().strftime("%Y-%m-%d
%H:%M:%S")

        # 1. Simulate reading
        temperature = simulate_temperature_reading()

        # 2. Validate reading
        is_valid, status_message = is_reading_valid(temperature, min_safe_temp,
max_safe_temp)

        # 3. Format log entry
        if not is_valid:
            log_status = f"INVALID_{status_message}" # E.g.,
INVALID_HIGH_ALERT
        else:
            log_status = status_message # E.g., NORMAL

        log_entry = format_log_entry(current_timestamp, sensor_id, temperature,
log_status)

        # 4. Write to log
        write_to_log(log_entry)

        time.sleep(interval_seconds) # Wait for the next logging interval

    print(f"--- Temperature Logger for {sensor_id} Finished ---")

# --- Lab 3: Modular Temperature Logger ---
# Uncomment the line below to run Lab Exercise 3
# run_temperature_logger(sensor_id="THERM_001", duration_seconds=10,
interval_seconds=1.5)

```

4.2 Challenge/Extension Ideas for Lab 3




- **Logging Levels:** Enhance `format_log_entry` and `write_to_log` to support different logging levels (e.g., DEBUG, INFO, WARNING, ERROR) and filter which levels are printed/written.
- **External Configuration:** Read `sensor_id`, `duration_seconds`, `interval_seconds`, `MIN_SAFE_TEMP_C`, `MAX_SAFE_TEMP_C` from a JSON or YAML configuration file using standard library modules (`json`, `yaml` if installed).
- **Error Handling (File I/O):** If `write_to_log` were writing to an actual file, add `try-except` blocks to handle potential IOError (e.g., disk full, permissions issue).
- **Multi-Sensor Logging:** Adapt `run_temperature_logger` to take a list of sensor IDs and their respective safe ranges, then log data for all sensors concurrently (conceptually, perhaps using threading/asyncio - hint for Day 5).
- **Data Aggregation:** Implement a module that calculates a rolling average or identifies peak/trough temperatures over a certain window of readings before logging.

5. Key Takeaways

This hands-on lab session provided practical experience in applying the advanced Python concepts learned throughout Day 2, solidifying your understanding and demonstrating their utility in real-world engineering scenarios.

- **Decorator Application:** Gained practical experience in creating and applying decorators to add cross-cutting concerns like timing functions without modifying their core logic.
- **Closure for Adaptive Behavior:** Implemented closures to create specialized functions that "remember" their initial configuration, enabling adaptive behavior and statefulness without relying on classes or global variables.
- **Modular Design in Practice:** Applied the Single Responsibility Principle to break down a complex logging system into smaller, manageable, and reusable functions, improving code organization and maintainability.
- **Integration of Concepts:** Saw how various Python features (functions, loops, conditionals, type hints, standard library modules) work together to build a functional application.
- **Problem-Solving Through Implementation:** Reinforced

ISRO URSC – Python Training | Analog Data | 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.
