

Day 2 Session 2:

Advanced Function Concepts - Deep Dive

This notebook delves into more advanced concepts related to Python functions, building on the foundation from Session 1. We will explore closures, decorators (a powerful application of first-class functions), and utilities from the **functools** module that enhance function behavior and performance. These concepts are crucial for writing elegant, reusable, and maintainable code in complex engineering applications.

1. First-Class Functions

Before diving into closures and decorators, it's essential to understand that functions in Python are first-class objects (or "first-class citizens"). This means:

- A function can be assigned to a variable.
- A function can be passed as an argument to another function.
- A function can be returned as the result of another function.
- A function can be stored in data structures (lists, dictionaries).

This characteristic is the foundation for functional programming paradigms and for creating powerful constructs like decorators.

1.1 Assigning Functions to Variables

```
Python
def log_message(msg):
    """Logs a message with a prefix."""
    print(f"INFO: {msg}")

# Assign the function to a variable
my_logger = log_message

# Call the function using the variable
my_logger("This is a log message via a variable.")
```

1.2 Passing Functions as Arguments (Higher-Order Functions)

A function that takes one or more functions as arguments, or returns a function as its result, is called a higher-order function.

Python

```
def process_data(data, strategy_function):
    """
    Processes data using a given strategy function.
    `strategy_function` is a function passed as an argument.
    """
    print(f"Original data: {data}")
    processed_data = strategy_function(data)
    print(f"Processed data: {processed_data}")
    return processed_data

def multiply_by_two(values):
    """Strategy: multiplies all values in a list by two."""
    return [v * 2 for v in values]

def filter_negatives(values):
    """Strategy: filters out negative values from a list."""
    return [v for v in values if v >= 0]

sensor_readings = [10, 20, -5, 15, -10]

print("--- Applying Multiply Strategy ---")
process_data(sensor_readings, multiply_by_two)

print("\n--- Applying Filter Strategy ---")
process_data(sensor_readings, filter_negatives)
```

1.3 Returning Functions from Functions

This concept is crucial for closures and decorators.

Python

```
def create_validator(min_val, max_val):
    """
    Returns a function that validates if a value is within a specified range.
    The returned function "remembers" min_val and max_val.
    """
    def validator_function(value):
        return min_val <= value <= max_val

    return validator_function

# Create specific validators
is_valid_temperature = create_validator(0, 100)    # Celsius
is_valid_pressure = create_validator(90, 110)     # kPa

print(f"Is 25°C a valid temperature? {is_valid_temperature(25)}")
print(f"Is 120°C a valid temperature? {is_valid_temperature(120)}")
print(f"Is 95 kPa a valid pressure? {is_valid_pressure(95)}")
print(f"Is 80 kPa a valid pressure? {is_valid_pressure(80)}")
```

2. Closures - Capturing Local Context

A **closure** is a function object that remembers values from its enclosing lexical scope even if the enclosing scope is no longer present. It allows a function to "close over" variables from its definition environment. This is a powerful feature for creating factory functions or functions with persistent state.

Closures occur when:

1. There is a nested function (a function defined inside another function).
2. The nested function refers to a non-global variable defined in its enclosing function.
3. The enclosing function returns the nested function.

Python

```
# Scenario: Creating specialized sensor data loggers with a fixed sensor ID

def create_sensor_logger(sensor_id):
```

```

"""
Returns a logger function for a specific sensor.
The returned logger 'closes over' the `sensor_id`.
"""

def log_reading(reading, unit):
    import datetime # Import inside to avoid circular dependency if this
was a module
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    print(f"[{timestamp}] Sensor {sensor_id}: {reading:.2f} {unit}")

    return log_reading

# Create loggers for different sensors
logger_temp = create_sensor_logger("Temp_A01")
logger_pressure = create_sensor_logger("Pres_B02")

# Use the specialized loggers
logger_temp(28.5, "°C")
logger_pressure(101.3, "kPa")
logger_temp(29.1, "°C") # Temp_A01 logger still remembers its ID

# Examining the closure
# You can inspect the closure's free variables (variables from enclosing scope)
print(f"\nLogger Temp closure vars:
{logger_temp.__closure__[0].cell_contents}")
print(f"Logger Pressure closure vars:
{logger_pressure.__closure__[0].cell_contents}")

```

Use Cases for Closures in Engineering/Science:

- **Factory Functions:** Creating functions with pre-configured parameters (e.g., a function to generate plots with specific titles/labels).
- **Stateful Functions:** Functions that maintain a "state" between calls without using global variables or class instances.
- **Callbacks:** Functions that need to remember context when executed later (e.g., event handlers).

3. Decorators: Writing and Applying

A **decorator** is a higher-order function that takes another function as an argument, extends its behavior without explicitly modifying it, and returns the enhanced function. Decorators provide a clean and reusable way to add "boilerplate" or cross-cutting concerns (like logging, timing, authentication, input validation) to multiple functions.

The **@decorator_name** syntax is syntactic sugar for **function = decorator_name(function)**.

3.1 Basic Decorator Structure

```
Python
def simple_decorator(func):
    """
    A simple decorator that prints messages before and after function
    execution.
    """
    def wrapper(*args, **kwargs):
        print(f"DEBUG: Entering function '{func.__name__}'...")
        result = func(*args, **kwargs) # Call the original function
        print(f"DEBUG: Exiting function '{func.__name__}'..")
        return result

    return wrapper

@simple_decorator
def perform_calculation(x, y):
    """Performs a simple calculation."""
    print(f" Performing calculation: {x} + {y}")
    return x + y

@simple_decorator
def fetch_sensor_data(sensor_id):
    """Simulates fetching data for a sensor."""
    print(f" Fetching data for {sensor_id}...")
    return {"id": sensor_id, "value": 25.5, "unit": "C"}

print("--- Applying Simple Decorator ---")
result_calc = perform_calculation(10, 20)
```

```

print(f"Calculation Result: {result_calc}")

sensor_data = fetch_sensor_data("Temp_001")
print(f"Sensor Data: {sensor_data}")

```

3.2 Decorators with Arguments

Decorators themselves can accept arguments. This requires an extra layer of nesting.

Python

```

def authorization_required(roles):
    """
    Decorator factory that takes roles as arguments.
    Returns the actual decorator.
    """
    def decorator(func):
        def wrapper(user_role, *args, **kwargs):
            if user_role in roles:
                print(f"INFO: User with role '{user_role}' authorized for '{func.__name__}'.")
                return func(user_role, *args, **kwargs)
            else:
                print(f"ACCESS DENIED: User with role '{user_role}' not authorized for '{func.__name__}'. "
                      f"Required roles: {roles}.")
                return None # Or raise an exception
        return wrapper
    return decorator

@authorization_required(roles=["admin", "operator"])
def calibrate_system(user_role, device_id):
    """Calibrates a system."""
    print(f"  Calibrating device {device_id} by {user_role}...")
    return f"Device {device_id} calibrated."

@authorization_required(roles=["admin"])
def delete_critical_data(user_role, data_id):
    """Deletes critical data (admin only)."""
    print(f"  Deleting critical data {data_id} by {user_role}...")

```

```
return f"Data {data_id} deleted."
```

```
print("\n--- Applying Decorators with Arguments (Authorization) ---")
calibrate_system("operator", "XYZ-001")
calibrate_system("guest", "XYZ-002") # Denied
delete_critical_data("admin", "LOG-DATA-999")
delete_critical_data("operator", "LOG-DATA-1000") # Denied
```

3.3 Common Use Cases for Decorators

Decorators are highly versatile for cross-cutting concerns:

- **Logging:** Automatically log function calls, arguments, and return values.
- **Timing/Performance Measurement:** Measure execution time of functions.
- **Input Validation:** Validate function arguments before execution.
- **Authentication/Authorization:** Restrict access to functions based on user roles.
- **Caching:** Store results of expensive function calls (e.g., `functools.lru_cache`).
- **Retries:** Automatically retry a function call if it fails (e.g., network operations).

Summary of Decorator Use Cases

Use Case	Description	Example Output
Logging	Prints information about function calls (e.g., arguments, results).	LOG: Calling function 'my_func' with args...
Timing	Measures and prints the execution time of a function.	Function 'my_func' executed in 0.05s.
Input Validation	Checks function arguments against predefined rules.	ERROR: Invalid input for 'param_x'.
Authorization	Restricts function access based on user permissions.	ACCESS DENIED.

Caching	Stores and reuses results of expensive function calls.	Returning cached result for 'key'.
Retries	Automatically retries a function call on failure.	Attempt 1 failed. Retrying...

3.4 The @wraps Decorator

When you use a decorator, the **wrapper** function effectively replaces the original function. This means the `__name__`, `__doc__`, and other metadata of the original function are lost. The `@functools.wraps` decorator solves this by copying the original function's metadata to the **wrapper** function.

```
Python
import functools

def timed_execution(func):
    """
    Decorator to measure the execution time of a function.
    """
    @functools.wraps(func) # This line copies metadata from 'func' to
    'wrapper'
    def wrapper(*args, **kwargs):
        import time # Import inside to keep scope clean
        start_time = time.perf_counter()
        result = func(*args, **kwargs)
        end_time = time.perf_counter()
        execution_time = end_time - start_time
        print(f"INFO: Function '{func.__name__}' executed in
        {execution_time:.4f} seconds.")
        return result

    return wrapper

@timed_execution
def long_running_simulation(iterations):
    """
    Simulates a long-running numerical simulation.
    """
    total = 0
    for i in range(iterations):
        total += i * 0.0001 # A small calculation
```



```

    return total

print("--- Using @wraps ---")
long_running_simulation(1000000)
print(f"Docstring of long_running_simulation: {long_running_simulation.__doc__}")
print(f"Name of long_running_simulation: {long_running_simulation.__name__}")

# Without @wraps, __doc__ and __name__ would belong to the 'wrapper' function

```

3.5 Class-Based Decorators

Decorators can also be implemented as classes. A class becomes a decorator if its instances are callable (by implementing the `__call__` method) and it can take a function as an argument in its `__init__` method. This is useful when the decorator needs to maintain state across multiple calls to the decorated function.

```

Python
import functools

class RetryDecorator:
    """
    A class-based decorator to retry a function call N times on failure.
    """
    def __init__(self, max_retries=3, delay_sec=1):
        self.max_retries = max_retries
        self.delay_sec = delay_sec
        # We use functools.wraps here too for class-based decorators
        # if we want to preserve metadata, usually on the __call__ method

    def __call__(self, func):
        @functools.wraps(func) # Apply wraps to the actual callable wrapper
        def wrapper(*args, **kwargs):
            import time
            for attempt in range(1, self.max_retries + 1):
                try:
                    print(f"Attempt {attempt}/{self.max_retries} for {func.__name__}...")
                    result = func(*args, **kwargs)
                    print(f"'{func.__name__}' succeeded on attempt {attempt}.")

```

```

        return result
    except Exception as e:
        print(f"'{func.__name__}' failed on attempt {attempt}:
{e}")

        if attempt < self.max_retries:
            time.sleep(self.delay_sec)
            self.delay_sec *= 2 # Exponential backoff for example
        else:
            print(f"'{func.__name__}' failed after
{self.max_retries} attempts.")
            raise # Re-raise the last exception

    return wrapper

@RetryDecorator(max_retries=3, delay_sec=0.5)
def unstable_network_call(data_packet):
    """Simulates an unstable network call that fails sometimes."""
    import random
    if random.random() < 0.7: # 70% chance of failure
        raise ConnectionError("Network connection unstable.")
    print(f"Successfully sent data packet: {data_packet}")
    return "Data sent"

print("\n--- Applying Class-Based Decorator (Retry) ---")
try:
    unstable_network_call("Telemetry_Packet_A")
    unstable_network_call("Calibration_Request_B")
except ConnectionError as e:
    print(f"Caught top-level error: {e}")

```

4. functools: lru_cache, singledispatch

The functools module provides higher-order functions and operations on callable objects, often used with decorators.

4.1 functools.lru_cache (Least Recently Used Cache)

A decorator that caches the results of a function call. If the function is called again with the same arguments, the cached result is returned instead of re-executing the function. This is extremely useful for optimizing expensive or frequently called functions.

`lru_cache` supports a `maxsize` argument to limit the number of entries in the cache. When the cache is full, the least recently used entry is discarded.

```
Python
import functools
import time

@functools.lru_cache(maxsize=128) # Cache up to 128 unique calls
def fetch_complex_calculation(input_param):
    """
    Simulates a complex, time-consuming calculation or data retrieval.
    """
    print(f"Performing complex calculation for {input_param}...")
    time.sleep(0.5) # Simulate delay
    result = input_param * 123.456 + (input_param % 7)
    return result

print("--- Using functools.lru_cache ---")
print(f"Result 1: {fetch_complex_calculation(10)}") # Calculated
print(f"Result 2: {fetch_complex_calculation(20)}") # Calculated
print(f"Result 3: {fetch_complex_calculation(10)}") # Cached
print(f"Result 4: {fetch_complex_calculation(30)}") # Calculated
print(f"Result 5: {fetch_complex_calculation(20)}") # Cached

# Inspect cache statistics
print(f"\nCache Info: {fetch_complex_calculation.cache_info()}")

# Clear the cache
fetch_complex_calculation.cache_clear()
print(f"Cache Info after clear: {fetch_complex_calculation.cache_info()}")
print(f"Result 6: {fetch_complex_calculation(10)}") # Recalculated after clear
```

Use Cases for `lru_cache` in Engineering/Science:

- Memoizing results of expensive numerical simulations with repetitive inputs.
- Caching database queries or API calls for frequently accessed data (be mindful of data staleness).
- Optimizing recursive functions (e.g., Fibonacci sequence, dynamic programming problems).

4.2 `functools.singledispatch`

A decorator that transforms a function into a **single-dispatch** generic function. This means it can behave differently based on the type of its first argument. It's an alternative to traditional method overloading (which Python doesn't have by default for functions) and allows for flexible extensibility.

Python

```
from functools import singledispatch
from typing import Union, List

@singledispatch
def process_sensor_input(data_input):
    """
    Generic function to process various types of sensor input.
    Default implementation.
    """
    print(f"Processing unknown sensor input type: {type(data_input).__name__}
    -> {data_input}")

@process_sensor_input.register(int)
@process_sensor_input.register(float)
def _(data_input: Union[int, float]):
    """Registers for int and float types. Converts to string with units."""
    print(f"Processing numerical sensor value: {data_input:.2f} units (Type:
    {type(data_input).__name__})")

@process_sensor_input.register(str)
def _(data_input: str):
    """Registers for string type. Parses as a status message."""
    print(f"Processing string status message: '{data_input.upper()}' (Type:
    {type(data_input).__name__})")

@process_sensor_input.register(list)
def _(data_input: List):
    """Registers for list type. Processes as a batch of readings."""
    if all(isinstance(x, (int, float)) for x in data_input):
        avg = sum(data_input) / len(data_input) if data_input else 0
        print(f"Processing batch of readings (List): Average = {avg:.2f}")
    else:
        print(f"Processing mixed list: {data_input}")
```

```

print("--- Using functools.singledispatch ---")
process_sensor_input(100)           # Calls int/float version
process_sensor_input(99.5)          # Calls int/float version
process_sensor_input("OPERATIONAL") # Calls str version
process_sensor_input([10, 20, 30])  # Calls list version (numerical)
process_sensor_input([1, "error", 3]) # Calls list version (mixed)
process_sensor_input({"id": "XYZ"})  # Calls default version (dict not
                                     registered)

```

Use Cases for `singledispatch` in Engineering/Science:

- Creating a common interface for processing different types of sensor payloads (e.g., raw binary, parsed JSON, numerical arrays).
- Implementing polymorphic functions where behavior depends on the concrete type of the first argument, especially useful for extensible APIs.
- Defining different serialization/deserialization logic based on object types.

5. Key Takeaways

This session explored advanced functional programming concepts in Python, providing powerful tools for writing more modular, extensible, and performant code.




- **First-Class Functions:** Understood that functions are objects and can be treated like any other data type, which is foundational for higher-order functions.
- **Closures:** Learned how nested functions can "remember" and access variables from their enclosing scope, even after the outer function has finished execution, enabling functions with persistent state.
- **Decorators:** Mastered the concept of decorators as a powerful design pattern to extend or modify the behavior of functions without altering their source code, using the `@` syntax for readability. Explored common use cases like logging, timing, and authorization, and understood the importance of `@functools.wraps`.
- **Class-Based Decorators:** Gained insight into implementing decorators using classes, particularly useful when state needs to be managed across multiple calls or across instances of the decorated function.
- **`functools.lru_cache`:** Discovered how to easily cache function results for

performance optimization, especially for expensive or frequently repeated computations.

- **functools singledispatch:** Understood how to create generic functions that dispatch to different implementations based on the type of the first argument, enabling flexible and extensible type-based polymorphism.

These advanced function concepts provide a significant leap in your ability to write sophisticated and maintainable Python applications for complex engineering and scientific challenges.

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.
