Day 2 - Session 1:
Python Functions - Deep Dive

This notebook delves into Python functions, which are fundamental building blocks for organizing and reusing code. We'll explore their definition, various argument passing mechanisms, flexible argument handling, documentation, type hinting for clarity, and the powerful concept of functions as first-class objects for functional programming.

1. Defining, Calling, and Returning Values

Functions are blocks of organized, reusable code that perform a single, related action. They help break down programs into smaller, manageable, and modular pieces, making code easier to understand, maintain, and debug.

1.1 Defining and Calling Functions

Functions are defined using the `def` keyword, followed by the function name, parentheses `()`, and a colon `:`. The function body is indented.

Python
```python
# Defining a simple function to log an event
def log_event(event_message):
    """
    Logs a simple event message to the console.
    This is a docstring, explaining what the function does.
    """
    print(f"LOG: {event_message}")

# Calling the function
log_event("System initialization started.")
log_event("Sensor A reading complete")
```

1.2 Returning Values

Functions can optionally return a value using the `return` keyword. If `return` is not explicitly used, the function implicitly returns `None`.

Python

```python
# Function to calculate delta between two sensor readings
def calculate_delta(reading1, reading2):
    """
    Calculates the absolute difference between two sensor readings.
    Returns the absolute delta.
    """
    delta = abs(reading1 - reading2)
    return delta

# Calling the function and storing the returned value
current_temp = 25.0
previous_temp = 23.5
temperature_change = calculate_delta(current_temp, previous_temp)
print(f"Temperature change: {temperature_change:.2f}°C")

pressure_current = 101.2
pressure_previous = 100.0
pressure_change = calculate_delta(pressure_current, pressure_previous)
print(f"Pressure change: {pressure_change:.2f} kPa")
```

Python

```python
# Functions can return multiple values as a tuple
def get_sensor_statistics(data_points):
    """
    Calculates the minimum, maximum, and average of a list of data points.
    Returns these three values as a tuple.
    """
    if not data_points:
        return None, None, None # Return None for all if list is empty
    min_val = min(data_points)
    max_val = max(data_points)
    avg_val = sum(data_points) / len(data_points)
    return min_val, max_val, avg_val

sensor_readings = [12.5, 13.1, 11.9, 14.0, 12.8]
min_r, max_r, avg_r = get_sensor_statistics(sensor_readings) # Unpacking the
tuple
print(f"Sensor Statistics: Min={min_r}, Max={max_r}, Avg={avg_r:.2f}")

empty_readings = []
min_e, max_e, avg_e = get_sensor_statistics(empty_readings)
print(f"Empty Readings Statistics: Min={min_e}, Max={max_e}, Avg={avg_e}")
```

2. Positional vs. Keyword vs. Default Arguments

Python offers flexible ways to pass arguments to functions, enhancing readability and making functions more versatile.

2.1 Positional Arguments

Arguments are passed to parameters in the order they are defined in the function signature. The position matters.

```python
def generate_report(project_id, author_name, report_date):
    """
    Generates a basic project report header.
    """
    print(f"Project ID: {project_id}")
    print(f"Author: {author_name}")
    print(f"Date: {report_date}")

# Calling with positional arguments
generate_report("ISRO-SAT-001", "ABCDEF", "")
```

2.2 Keyword Arguments

Arguments are passed by explicitly naming the parameter they correspond to, using `param_name=value`. The order does not matter as long as the names are correct. This improves readability, especially for functions with many parameters.

```python

# Using the same function
generate_report(report_date="2025-06-21", project_id="URSC-TEST-002",
author_name="Er. B. Singh")

# You can mix positional and keyword arguments, but positional must come first
generate_report("TEL-ANL-003", report_date="2025-06-22", author_name="Ms. C.
DDEF")
```

```
# generate_report(report_date="2025-06-22", "TEL-ANL-003", author_name="Ms. C.
Devi") # This would be an error
```

2.3 Default Arguments

Parameters can have default values. If an argument is not provided for such a parameter during a function call, its default value is used. Default arguments must come after any non-default (positional-only or positional/keyword) arguments.

```python
def configure_sensor(sensor_id, sampling_rate_hz=10, calibration_offset=0.0,
debug_mode=False):
    """
    Configures a sensor with optional parameters.
    """
    print(f"\nConfiguring Sensor: {sensor_id}")
    print(f"  Sampling Rate: {sampling_rate_hz} Hz")
    print(f"  Calibration Offset: {calibration_offset}")
    print(f"  Debug Mode: {debug_mode}")

# Using all default arguments for optional parameters
configure_sensor("TempSensor_001")

# Overriding some default arguments using keywords
configure_sensor("PressureSensor_002", sampling_rate_hz=50, debug_mode=True)

# Overriding a specific default argument using position (less readable for
optional args)
configure_sensor("FlowMeter_003", 20) # 20 is for sampling_rate_hz

# Combining all types: Positional, then Keyword defaults
configure_sensor("VoltMeter_004", sampling_rate_hz=5, calibration_offset=0.15)
```

Important Note on Mutable Default Arguments:

Avoid using mutable objects (like lists or dictionaries) as default argument values. They are created once when the function is defined, and subsequent calls to the function will modify the same mutable object, leading to unexpected behavior.

```python
def add_to_list_BAD(item, my_list=[]): # BAD PRACTICE
    my_list.append(item)
    return my_list

print(add_to_list_BAD(1)) # Output:
print(add_to_list_BAD(2)) # Output: - Unexpected! `my_list` is the same object
print(add_to_list_BAD(3)) # Output: - This works as a new list is passed

def add_to_list_GOOD(item, my_list=None): # GOOD PRACTICE
    if my_list is None:
        my_list = []
    my_list.append(item)
    return my_list

print(add_to_list_GOOD(1)) # Output:
print(add_to_list_GOOD(2)) # Output: - Correct, new list each time
```

Summary of Argument Types

Type	Syntax	Order Important?	Readability	Flexibility
Positional	`func(val1, val2)`	Yes	Hig...	Low
Keyword	`func(p1=val1, p2=val2)`	No	High	High

Default	`def func(p=default_val):`	Yes (must be after non-defaults)	High	High

3. `*args` and `**kwargs` for Flexible APIs

These special syntaxes allow functions to accept an arbitrary number of positional arguments (`*args`) or keyword arguments (`**kwargs`), making functions highly flexible and adaptable to varying inputs.

3.1 `*args` (Arbitrary Positional Arguments)

The `*args` parameter collects all unmatched positional arguments into a tuple. The name `args` is a convention; you can use any name prefixed with `*`.

Python

```python
def calculate_average_readings(*readings):
    """
    Calculates the average of an arbitrary number of sensor readings.
    'readings' will be a tuple of all positional arguments passed.
    """
    if not readings:
        return 0.0
    return sum(readings) / len(readings)

print(f"Avg of (10, 12, 11): {calculate_average_readings(10, 12, 11):.2f}")
print(f"Avg of (15.5, 16.0, 14.8, 17.2): {calculate_average_readings(15.5,
16.0, 14.8, 17.2):.2f}")
print(f"Avg of no readings: {calculate_average_readings():.2f}")
```

```python
# You can combine positional arguments with *args
def process_data_stream(stream_name, *data_points, unit="units"):
    """
    Processes a named data stream with multiple data points.
    """
    print(f"\nProcessing stream: {stream_name}")
    if data_points:
        avg = sum(data_points) / len(data_points)
          print(f"  Received {len(data_points)} data points. Average: {avg:.2f}
{unit}")
    else:
        print("  No data points received.")

process_data_stream("Telemetry_Channel_A", 100, 102, 99, 105)
process_data_stream("Sensor_Status_Log", "OK", "ERROR", "OK") # Can take mixed
types, but sum() would fail
process_data_stream("Empty_Stream")
process_data_stream("Pressure_Stream", 50.1, 50.2, 50.3, unit="kPa")
```

3.2 `**kwargs` (Arbitrary Keyword Arguments)

The `**kwargs` parameter collects all unmatched keyword arguments into a dictionary. The name kwargs is a convention; you can use any name prefixed with `**`.

```python

def log_sensor_event(event_type, **details):
    """
    Logs a sensor event with a type and arbitrary key-value details.
    'details' will be a dictionary of all keyword arguments passed.
    """
    print(f"\n--- Sensor Event: {event_type.upper()} ---")
    for key, value in details.items():
        print(f"  {key.replace('_', ' ').title()}: {value}")
    print("--------------------------")

log_sensor_event("Temperature Exceeds Threshold",
                 sensor_id="TS_007",
                 current_temp=38.5,
                 threshold=35.0,
                 unit="Celsius",
                 priority="High")

log_sensor_event("Device Status Update",
                 device_name="Actuator_Hydraulic",
                 status="Operational",
                 uptime_hours=125.7)

# No extra keyword arguments
log_sensor_event("System Idle")
```

3.3 Order of Arguments in Function Signature

When combining different argument types, Python enforces a specific order:

1. Positional-only arguments (rare, used with `/`)
2. Positional or keyword arguments
3. **`*args`**
4. Keyword-only arguments (used with `*` or `*, param_name`)
5. **`**kwargs`**

Python

```python
def flexible_api(param1, param2="default", *args, kw_only_param, **kwargs):
    print(f"param1: {param1}")
    print(f"param2 (default): {param2}")
    print(f"args: {args}")  # A tuple
    print(f"kw_only_param (keyword-only): {kw_only_param}")
    print(f"kwargs: {kwargs}")  # A dictionary

# Example call
flexible_api(10, "value2", 1, 2, 3, kw_only_param="required",
extra_info="test", id=123)
# Note: kw_only_param MUST be passed as a keyword argument because it's after
*args
```

Use Cases for *args and **kwargs:

- **Wrapper Functions/Decorators:** Passing arbitrary arguments to the wrapped function.
- **Flexible APIs:** Functions that need to handle varying numbers of inputs (e.g., a logging function that can take various details).
- **Inheritance:** Passing arguments from a child class's __init__ to a parent class's __init__.

4. Docstrings, Annotations, and Type Hints (Recap and Deep Dive for Functions)

In Day 1, Session 1, we introduced type hints generally. Here, we'll focus on their application specifically to function parameters and return values, and emphasize the importance of docstrings.

4.1 Docstrings

Docstrings are multi-line strings (enclosed in triple quotes """Docstring goes here.""") immediately following the function, method, class, or module definition. They provide a brief overview of what the code does. They are accessible via help() or .__doc__.

```
Python

def compute_orbital_velocity(radius_km: float, mass_of_body_kg: float) ->
float:
    """
    Calculates the orbital velocity of a satellite around a central body.

    This function uses the formula for circular orbital velocity:
    v = sqrt(G * M / r)
    where G is the gravitational constant, M is the mass of the central body,
    and r is the orbital radius.

    Args:
        radius_km (float): The orbital radius in kilometers.
        mass_of_body_kg (float): The mass of the central body in kilograms.
    Returns:
        float: The orbital velocity in kilometers per second (km/s).
    """
    GRAVITATIONAL_CONSTANT = 6.67430e-11 # N(m/kg)^2 or m^3/(kg s^2)
    radius_m = radius_km * 1000 # Convert km to meters

    # Velocity in m/s
    velocity_mps = (GRAVITATIONAL_CONSTANT * mass_of_body_kg / radius_m)**0.5
    return velocity_mps / 1000 # Convert m/s to km/s

# Accessing the docstring
print(compute_orbital_velocity.__doc__)

# Example Usage
earth_radius_km = 6371 # Average radius of Earth
earth_mass_kg = 5.972e24 # Mass of Earth
orbit_altitude_km = 400 # International Space Station altitude
orbital_radius_iss_km = earth_radius_km + orbit_altitude_km

iss_velocity = compute_orbital_velocity(orbital_radius_iss_km, earth_mass_kg)
print(f"\nISS orbital velocity: {iss_velocity:.2f} km/s")
```

Common Docstring Formats:

- **Sphinx (reStructuredText)**: Widely used in scientific Python libraries (e.g., NumPy, SciPy).
- **Google Style**: More concise, used often in Google projects.
- **NumPy Style**: A variation of Sphinx style, very popular for numerical libraries.
- **Epytext**: Less common now.

Consistency in docstring format is important for project maintainability.

4.2 Annotations and Type Hints for Functions

Type hints (introduced in PEP 484) specify the expected types of function parameters and the return value. They improve code clarity, enable static analysis by tools (like MyPy), and help IDEs provide better autocompletion and error checking.

Syntax: `parameter_name: Type` for parameters, `-> ReturnType` for return value.

```python
from typing import List, Tuple, Dict, Union, Optional


def analyze_sensor_readings(
    sensor_id: str,
    readings: List[float],
    threshold_info: Optional[Dict[str, float]] = None
) -> Tuple[float, float, str]:
    """
    Analyzes a list of sensor readings and determines a status.

    Args:
        sensor_id: Unique identifier for the sensor.
        readings: A list of floating-point sensor readings.
        threshold_info: Optional dictionary containing 'high' and 'low'
thresholds.

    Returns:
        A tuple containing: (min_reading, max_reading, status_message).
    """
    if not readings:
        return 0.0, 0.0, "No readings"

    min_val = min(readings)
    max_val = max(readings)
    status = "Normal"

    if threshold_info:
        high_threshold = threshold_info.get("high")
        low_threshold = threshold_info.get("low")

        if high_threshold is not None and max_val > high_threshold:
            status = "High Alert"
        elif low_threshold is not None and min_val < low_threshold:
            status = "Low Warning"

    return min_val, max_val, status


# Example usage with type hints
temp_data: List[float] = [22.5, 23.1, 21.9, 24.0, 22.8]
temp_thresholds: Dict[str, float] = {"high": 23.5, "low": 20.0}
```

```
min_t, max_t, status_t = analyze_sensor_readings("Temp001", temp_data,
temp_thresholds)
print(f"\nTemp Sensor Analysis: Min={min_t}, Max={max_t}, Status='{status_t}'")

pressure_data: List[float] = [100.1, 102.5, 98.7, 101.5]

# Calling without optional thresholds
min_p, max_p, status_p = analyze_sensor_readings("Pres002", pressure_data)
print(f"Pressure Sensor Analysis: Min={min_p}, Max={max_p},
Status='{status_p}'")
```

Benefits of Annotations/Type Hints:

- **Self-documenting code**: Makes the intent of the function clear.
- **Improved readability and maintainability**: Easier to understand function contracts.
- **Static analysis:** Tools like MyPy can catch type-related bugs before runtime.
- **IDE support**: Better autocompletion, refactoring, and error highlighting.

5. Writing Modular Functions

Modular programming is the practice of breaking down a large program into smaller, self-contained, and independent modules or functions. Each module/function performs a specific task and has a well-defined interface.

5.1 Principles of Modular Design

- **Single Responsibility Principle (SRP)**: Each function should have one, and only one, reason to change. It should do one thing well.
- **High Cohesion**: The elements within a function should be strongly related to its single purpose.
- **Loose Coupling**: Functions should be as independent as possible, minimizing their reliance on other parts of the code. This makes them easier to test, reuse, and maintain.
- **Readability**: Functions should be easy to understand at a glance.
- **Reusability**: Design functions to be generic enough to be used in different parts of your application or even different projects.

5.2 Example: Refactoring for Modularity (Telemetry Processing)

Consider a task of processing telemetry data, which involves reading, validating, calculating, and logging.

Bad (Monolithic) Approach:

```python
# This is a conceptual example of a monolithic function.
# In a real scenario, it would be much longer and harder to manage.

def process_telemetry_monolithic(raw_data_string):
    # 1. Parse data string
    parts = raw_data_string.strip().split(',')
    if len(parts) != 4:
        print("ERROR: Invalid data format.")
        return

    try:
        timestamp = parts[0]
        sensor_type = parts[1]
        value_str = parts[2]
        status_code_str = parts[3]

        # 2. Validate and Convert
        value = float(value_str)
        status_code = int(status_code_str)

        if value < 0 or value > 1000:
            print(f"WARNING: Value {value} out of range for {sensor_type}.")
            return

        # 3. Apply business logic/classification
        if status_code == 0:
            status_desc = "OK"
        elif status_code == 1:
            status_desc = "WARNING"
        elif status_code == 2:
            status_desc = "ERROR"
        else:
            status_desc = "UNKNOWN_STATUS"

        # 4. Log results
        print(f"[{timestamp}] Sensor: {sensor_type}, Value: {value:.2f}, 
Status: {status_desc}")

    except ValueError as e:
        print(f"ERROR: Data conversion failed: {e}")


print("\n--- Monolithic Telemetry Processing ---")
process_telemetry_monolithic("2025-06-21T10:00:00,Temp,25.5,0")
process_telemetry_monolithic("2025-06-21T10:00:01,Pres,101.2,0")
process_telemetry_monolithic("2025-06-21T10:00:02,Hum,1050,1")       # Out of
range value
process_telemetry_monolithic("2025-06-21T10:00:03,Volt,invalid,0")   # Invalid
data type
```

Good (Modular) Approach:

Python
```python
def parse_telemetry_string(raw_data_string: str) -> Optional[Dict[str, str]]:
    """Parses a raw telemetry string into a dictionary of string parts."""
    parts = raw_data_string.strip().split(',')
    if len(parts) != 4:
        print(f"ERROR: Invalid format for raw data: '{raw_data_string}'
(Expected 4 parts, got {len(parts)})")
        return None

    return {
        "timestamp": parts[0],
        "sensor_type": parts[1],
        "value_str": parts[2],
        "status_code_str": parts[3]
    }


def validate_and_convert_telemetry(parsed_data: Dict[str, str]) ->
Optional[Dict[str, Union[str, float, int]]]:
    """Validates and converts telemetry data types."""
    try:
        value = float(parsed_data["value_str"])
        status_code = int(parsed_data["status_code_str"])

        if not (0 <= value <= 1000):  # Assuming a general valid range for
sensor values
            print(f"WARNING: Value {value} out of defined range for
{parsed_data['sensor_type']}.")
```

```python
            return None  # Indicate invalid conversion

        return {
            "timestamp": parsed_data["timestamp"],
            "sensor_type": parsed_data["sensor_type"],
            "value": value,
            "status_code": status_code
        }

    except ValueError as e:
        print(f"ERROR: Data conversion failed for {parsed_data}: {e}")
        return None


def classify_telemetry_status(status_code: int) -> str:
    """Classifies a numerical status code into a human-readable string."""
    if status_code == 0:
        return "OK"
    elif status_code == 1:
        return "WARNING"
    elif status_code == 2:
        return "ERROR"
    else:
        return "UNKNOWN_STATUS"


def log_processed_telemetry(processed_data: Dict[str, Union[str, float, int]],
status_desc: str) -> None:
    """Logs the final processed telemetry data."""
    print(f"[{processed_data['timestamp']}] Sensor:
{processed_data['sensor_type']}, "
          f"Value: {processed_data['value']:.2f}, Status: {status_desc}")


def process_telemetry_modular(raw_data_string: str) -> None:
    """Orchestrates the modular processing of a single telemetry data
string."""
    parsed = parse_telemetry_string(raw_data_string)
    if parsed is None:
        return  # Stop if parsing failed

    converted = validate_and_convert_telemetry(parsed)
    if converted is None:
        return  # Stop if validation/conversion failed

    status_desc = classify_telemetry_status(converted["status_code"])
    log_processed_telemetry(converted, status_desc)
```

```
print("\n--- Modular Telemetry Processing ---")
process_telemetry_modular("2025-06-21T10:00:00,Temp,25.5,0")
process_telemetry_modular("2025-06-21T10:00:01,Pres,101.2,0")
process_telemetry_modular("2025-06-21T10:00:02,Hum,1050,1")       # Out of
range value handled by validation
process_telemetry_modular("2025-06-21T10:00:03,Volt,invalid,0")   # Invalid
data type handled by conversion
process_telemetry_modular("2025-06-21T10:00:04,Flow,12.3")        # Incorrect
number of parts
```

Benefits of Modular Functions:

- **Testability**: Each small function can be tested independently.
- **Maintainability**: Easier to find and fix bugs, or update specific logic without affecting other parts.
- **Reusability**: Functions like `classify_telemetry_status` can be reused elsewhere.
- **Readability**: The `process_telemetry_modular` function acts as an orchestra, showing the flow of logic without getting bogged down in details.

6. Functions as Data: `map`, `filter`, `reduce`, `functools.partial`

In Python, functions are "first-class objects," meaning they can be treated like any other variable: assigned to variables, passed as arguments to other functions, and returned from functions. This enables powerful functional programming paradigms.

6.1 `map(function, iterable)`

Applies a given function to each item of an iterable (e.g., a list) and returns an iterator of the results.

Python
```
# Scenario: Convert a list of temperatures from Celsius to Fahrenheit
```

```
def celsius_to_fahrenheit(celsius_temp: float) -> float:
    """Converts Celsius to Fahrenheit."""
    return (celsius_temp * 9 / 5) + 32


celsius_readings = [20.0, 25.0, 30.0, 35.0]

# Using a for loop (traditional)
fahrenheit_for_loop = []
for temp_c in celsius_readings:
    fahrenheit_for_loop.append(celsius_to_fahrenheit(temp_c))
print(f"Fahrenheit (for loop): {fahrenheit_for_loop}")

# Using map (functional approach)
fahrenheit_map = list(map(celsius_to_fahrenheit, celsius_readings))
print(f"Fahrenheit (map): {fahrenheit_map}")

# Applying a lambda function (anonymous function) with map
# Scenario: Square each element
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(f"Squared numbers: {squared_numbers}")
```

6.2 `filter(function, iterable)`

Constructs an iterator from elements of an iterable for which a function returns `True`.

Python
```
# Scenario: Filter out sensor readings below a minimum operational threshold

def is_operational(reading: float, min_threshold: float = 10.0) -> bool:
    """Checks if a sensor reading is above the minimum operational
threshold."""
    return reading >= min_threshold


all_readings = [8.5, 12.0, 9.9, 15.0, 7.0, 10.1]
MIN_OPERATIONAL_THRESHOLD = 10.0

# Using a for loop (traditional)
operational_for_loop = []
for reading in all_readings:
    if is_operational(reading, MIN_OPERATIONAL_THRESHOLD):
        operational_for_loop.append(reading)
print(f"Operational readings (for loop): {operational_for_loop}")

# Using filter (functional approach)
# Note: filter expects a function that returns a boolean
operational_filter = list(filter(lambda r: is_operational(r,
MIN_OPERATIONAL_THRESHOLD), all_readings))
print(f"Operational readings (filter): {operational_filter}")

# Filtering out error codes (using previous classification logic)
error_codes_present = [101, 0, 205, 999, 404]
error_code_info_map = {
    101: "Sensor calibration required.",
    205: "Power supply low.",
    404: "Software module not found."
}

def is_known_error(code: int, known_errors: Dict[int, str]) -> bool:
    """Checks if an error code is present in the known errors map."""
    return code in known_errors


known_errors_filter = list(filter(lambda code: is_known_error(code,
error_code_info_map), error_codes_present))
print(f"Known error codes in list: {known_errors_filter}")
```

6.3 `reduce(function, iterable[, initializer])`

Applies a rolling computation to sequential pairs of values in an iterable. It requires **`functools.reduce`**.

```python
from functools import reduce

# Scenario: Calculate the product of a list of values (e.g., gains in a signal
chain)

def multiply(x, y):
    """Multiplies two numbers."""
    return x * y
```

```
gains = [1.2, 1.5, 0.8, 1.1]

# Using a for loop (traditional)
total_gain_for_loop = 1
for gain in gains:
    total_gain_for_loop *= gain
print(f"Total gain (for loop): {total_gain_for_loop:.2f}")

# Using reduce (functional approach)
total_gain_reduce = reduce(multiply, gains)
print(f"Total gain (reduce): {total_gain_reduce:.2f}")

# Example: Sum of elements (reduce can do what sum() does)
sum_of_numbers = reduce(lambda x, y: x + y, [1, 2, 3, 4, 5])
print(f"Sum of numbers (reduce): {sum_of_numbers}")

# Using an initializer (starting value)
product_with_initial = reduce(multiply, [2, 3, 4], 10)  # 10 * 2 * 3 * 4
print(f"Product with initializer: {product_with_initial}")
```

6.4 `functools.partial`

Allows you to "freeze" some arguments of a function and generate a new function with a reduced number of arguments. This is useful for creating specialized versions of general functions.

Python
```python
from functools import partial
import datetime

# Scenario: Create specialized logging functions for different priority levels

def generic_log(level: str, message: str, timestamp: str) -> None:
    """A generic logging function."""
    print(f"[{timestamp}] {level.upper()}: {message}")


# Create specialized loggers using partial
info_logger = partial(
    generic_log,
    "INFO",
    timestamp=datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
)

warning_logger = partial(
    generic_log,
    "WARNING",
    timestamp=datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
)

error_logger = partial(
    generic_log,
    "ERROR",
    timestamp=datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
)

info_logger(message="Telemetry system online.")
warning_logger(message="Low disk space detected on log server.")
error_logger(message="Critical sensor failure in module B.")


# Scenario: Create a specialized filter for a specific threshold

def check_threshold(value: float, lower_bound: float, upper_bound: float) ->
bool:
    """Checks if a value is within a specified range."""
    return lower_bound <= value <= upper_bound


# Create a specific range checker for operational temperatures
is_operational_temp = partial(check_threshold, lower_bound=10.0,
upper_bound=30.0)

test_temps = [5.0, 15.0, 25.0, 35.0]
print("\nChecking operational temperatures:")
for temp in test_temps:
    status = "Operational" if is_operational_temp(temp) else "Out of Range"
    print(f"  Temperature {temp}°C: {status}")
```

7. Writing Modular Functions (Revisited and Best Practices)

To further emphasize the importance of modularity and reusability, here's a table summarizing best practices when designing your functions:

Best Practices for Modular Functions

Guideline	Description	Benefit
Single Responsibility	Each function should do one thing and do it well.	Easier to test, understand, and reuse.
Clear Naming	Function names should clearly describe their purpose.	Improves code readability and reduces cognitive load.
Docstrings	Provide a concise explanation of what the function does, its arguments, and what it returns.	Essential for documentation and understanding function usage.
Type Hints	Use type hints for parameters and return values.	Improves readability, enables static analysis, better IDE support.
Pure Functions (where possible)	Given the same inputs, always produce the same output, with no side effects.	Predictable, easy to test, can be optimized (memoization/caching).
Minimize Side Effects	If a function must have side effects, make it clear and limit them.	Reduces unexpected behavior and simplifies debugging.
Error Handling	Use try-except blocks for anticipated errors, or raise specific exceptions.	Makes functions robust and resilient to unexpected inputs/conditions.
Avoid Global State	Minimize reliance on global variables; pass necessary data as arguments.	Reduces coupling, improves testability, prevents unintended modifications.
Sensible Length	Functions should ideally be short, typically fitting on one screen.	Improves readability and helps adhere to SRP.
DRY (Don't Repeat Yourself)	Identify duplicated code snippets and refactor them into reusable functions.	Reduces code size, simplifies updates, prevents inconsistencies.

8. Key Takeaways

This session significantly deepened your understanding of Python functions, moving beyond basic definition to advanced usage crucial for scientific and engineering applications.

- **Function Fundamentals:** Solidified understanding of defining, calling, and returning single/multiple values from functions.
- **Flexible Argument Passing:** Mastered positional, keyword, and default arguments, enabling more readable and robust function interfaces.
- **Arbitrary Arguments:** Learned how *args and **kwargs allow functions to handle a varying number of inputs, making APIs highly adaptable.
- **Documentation & Clarity:** Reinforced the importance of docstrings and type hints for creating self-documenting and maintainable code, especially vital in collaborative engineering environments.
- **Modular Design Principles:** Explored the benefits of breaking down complex problems into smaller, testable, and reusable functions, improving overall code quality.
- **Functions as First-Class Objects:** Understood the powerful concept of treating functions as data, enabling functional programming paradigms with map, filter, reduce, and functools.partial for concise and expressive code.

These advanced function concepts are critical for writing scalable, maintainable, and efficient Python code for complex scientific and engineering projects.

ISRO URSC – Python Training | AnalogData | Rajath Kumar

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

---
