

Day 1 - Session 2:

Core Data Types & Operations - Deep Dive

This notebook delves into the fundamental data structures and operations in Python, crucial for efficient data handling in scientific and engineering contexts. We will explore strings, lists, tuples, sets, and dictionaries, along with powerful constructs like comprehensions and practical applications in data normalization and encoding.

1. Revisiting Python Environments (Quick Recap)

We introduced the critical concept of Python environments (venv and conda) for managing project dependencies and ensuring reproducibility. As a quick recap:

Summary of Environment Types

Environment Type	Primary Focus	Key Advantages	When to Use
venv (Virtual Environment)	Python package isolation	Lightweight, standard library, easy for Python-only projects.	General Python projects, web development where only Python packages are needed.
conda (Conda Environment)	Python versions & system-level (non-Python) packages	Comprehensive, robust for scientific stacks (handling C/Fortran dependencies), cross-language support.	Scientific computing, data science, machine learning, projects with complex binary dependencies.

For scientific and engineering applications, conda often provides a more seamless experience due to its ability to manage a wider range of dependencies. Regardless of your choice, using environments is a best practice.

2. Strings: Advanced Operations

Strings are fundamental for handling textual data, which is ubiquitous in engineering (e.g., parsing log files, instrument commands, metadata). Beyond basic concatenation, Python offers powerful methods for string manipulation.

2.1 F-Strings (Formatted String Literals - Python 3.6+)

F-strings provide a concise and readable way to embed expressions inside string literals. They are prefixed with `f` or `F`.

Python

Example: Displaying sensor readings with units

sensor_type = "Temperature"

reading_celsius = 25.734

unit = "°C"

Traditional string formatting (less readable)

print("Sensor Type: %s, Reading: %.2f %s" % (sensor_type, reading_celsius, unit))

print("Sensor Type: {}, Reading: {:.2f} {}".format(sensor_type, reading_celsius, unit))

F-string (most readable and concise)

print(f"Sensor Type: {sensor_type}, Reading: {reading_celsius:.2f} {unit}")

Embed expressions directly

threshold = 25.0

status = "OK" if reading_celsius <= threshold else "High"

print(f"Sensor Reading: {reading_celsius:.2f}{unit}, Status: {status}")

Calculations inside f-strings

pressure_psi = 14.7

pressure_pa = pressure_psi * 6894.76

print(f"Pressure: {pressure_psi:.1f} PSI (approx. {pressure_pa/1000:.2f} kPa)")

2.2 String Formatting and Methods

Python strings have a rich set of built-in methods for manipulation.

Common String Methods

Method	Description	Example	Output
<code>`upper()`</code>	Converts string to uppercase.	<code>`hello'.upper()`</code>	<code>"HELLO"</code> ▾
<code>`lower()`</code>	Converts string to lowercase.	<code>`"WORLD".lower()`</code>	<code>"world"</code> ▾
<code>`strip()`</code>	Removes leading/trailing whitespace.	<code>`" data ".strip()`</code>	<code>"data"</code> ▾
<code>`split(sep)`</code>	Splits string by separator into a list.	<code>`"A,B,C".split(',')`</code>	<code>['A', 'B', 'C']</code> ▾
<code>`join(iterable)`</code>	Joins elements of an iterable with the string as separator.	<code>`', '.join(['X','Y','Z'])`</code>	<code>"X,Y,Z"</code> ▾
<code>`replace(old, new)`</code>	Replaces occurrences of old with new.	<code>`"foo bar".replace('foo', 'baz')`</code>	<code>"baz bar"</code> ▾
<code>`startswith(prefix)`</code>	Checks if string starts with prefix.	<code>`"message".startswith('mess')`</code>	<code>"True"</code> ▾
<code>`endswith(suffix)`</code>	Checks if string ends with suffix.	<code>`"filename.txt".endswith('.txt')`</code>	<code>"True"</code> ▾
<code>`find(sub)`</code>	Returns lowest index of sub (or -1 if not found).	<code>`"abcde".find('c')`</code>	<code>"2"</code> ▾
<code>`count(sub)`</code>	Returns number of non-overlapping occurrences of sub.	<code>`"banana".count('na')`</code>	<code>"2"</code> ▾
<code>`isdigit()`</code>	Returns True if all chars are digits.	<code>`"123".isdigit()`</code>	<code>"True"</code> ▾
<code>`isalpha()`</code>	Returns True if all chars are alphabetic.	<code>`"Python".isalpha()`</code>	<code>"True"</code> ▾

Python

```
log_entry = " ERROR: Connection lost to Sensor_A at 2025-06-19 14:30:05 "  
  
# Clean and parse the log entry  
cleaned_entry = log_entry.strip()  
print(f"Cleaned entry: '{cleaned_entry}'")  
  
# Check for error type  
if cleaned_entry.startswith("ERROR"):  
    print("This is an error log.")  
  
# Extract components using split  
parts = cleaned_entry.split(':', 1) # Only split at the first ":"  
print(f"Parts: {parts}")  
log_level = parts[0]  
message = parts[1]  
print(f"Log Level: {log_level}, Message: {message}")  
  
# Replace a component  
formatted_message = message.replace("Sensor_A", "Telemetry_Unit_01")  
print(f"Formatted Message: {formatted_message}")  
  
# Check if a substring exists  
if "Connection lost" in formatted_message:  
    print("The connection was indeed lost.")  
  
# Joining elements to form a path  
path_components = ['data', 'sensors', 'temperature_log.csv']  
full_path = '/'.join(path_components)  
print(f"Full data path: {full_path}")
```

2.3 String Slicing

String slicing allows you to extract substrings using a syntax similar to list slicing: `[start:end:step]`.

- **start**: (Optional) The starting index (inclusive). Defaults to 0.
- **end**: (Optional) The ending index (exclusive). Defaults to the end of the string.
- **step**: (Optional) The step size. Defaults to 1. A negative step reverses the string.

Python

```
data_stream_id = "SC-2025-UNIT001-A-V1.2"

# Get the first two characters (Satellite Code)
satellite_code = data_stream_id[0:2] # or data_stream_id[:2]
print(f"Satellite Code: {satellite_code}")

# Get the year
year = data_stream_id[3:7]
print(f"Year: {year}")

# Get the unit number
unit_number = data_stream_id[8:11]
print(f"Unit Number: {unit_number}")

# Get the version (last 4 characters)
version = data_stream_id[-4:]
print(f"Version: {version}")

# Reverse the string
reversed_id = data_stream_id[::-1]
print(f"Reversed ID: {reversed_id}")

# Extract parts with negative indexing
# Example: UNIT001-A
specific_part = data_stream_id[-11:-4]
print(f"Specific Part (UNIT001-A): {specific_part}")
```

3. Core Data Types: Lists, Tuples, Sets, Dictionaries

These four built-in data types are the workhorses of data organization in Python. Understanding their characteristics and best use cases is crucial for efficient programming.

3.1 Lists

Lists are ordered, mutable sequences that can store items of different data types. They are defined by square brackets [].

List Characteristics

Characteristic	Description
Ordered	Elements have a defined order, which is maintained.
Mutable	Elements can be added, removed, or changed after creation.
Heterogeneous	Can contain elements of different data types.
Dynamic Size	Can grow or shrink as needed.
Indexed	Elements are accessed by integer indices (0-based).

Python

Creating lists

```
sensor_data_points = [10.5, 12.1, 11.8, 13.0, 9.7]
mixed_list = ["Sensor_X", 200, True, 15.6]
```

```
print(f"Sensor Data: {sensor_data_points}")
print(f"Mixed List: {mixed_list}")
```

Accessing elements (slicing works like strings)

```
print(f"First data point: {sensor_data_points[0]}")
```

```

print(f"Last data point: {sensor_data_points[-1]}")
print(f"Subset of data: {sensor_data_points[1:4]}") # Index 1, 2, 3

# Modifying lists (mutability)
sensor_data_points.append(10.1) # Add an element to the end
print(f"After append: {sensor_data_points}")

sensor_data_points[2] = 12.0 # Correctly change an element at index 2
print(f"After changing element at index 2: {sensor_data_points}")

sensor_data_points.insert(0, 9.5) # Insert at specific index
print(f"After insert: {sensor_data_points}")

sensor_data_points.pop() # Remove last element
print(f"After pop: {sensor_data_points}")

sensor_data_points.remove(12.1) # Remove by value (first occurrence)
print(f"After remove 12.1: {sensor_data_points}")

# Iterating
print("Iterating through sensor data:")
for data_point in sensor_data_points:
    print(data_point)

```

Use Cases for Lists:

- Storing a sequence of sensor readings over time.
- Collecting results from multiple experimental runs.
- Managing items in a queue or stack.
- Representing rows in a basic dataset.

3.2 Tuples

Tuples are ordered, immutable sequences. Once a tuple is created, its elements cannot be changed, added, or removed. They are defined by parentheses ().

Tuple Characteristics

Characteristic	Description
Ordered	Elements have a defined order, which is maintained.
Immutable	Elements cannot be changed after creation.
Heterogeneous	Can contain elements of different data types.
Fixed Size	Size cannot be changed after creation.
Indexed	Elements are accessed by integer indices (0-based).

Python

```
# Creating tuples
coordinates = (10.5, 20.3, 5.1) # (latitude, longitude, altitude)
sensor_reading_metadata = ("TempSensor_01", 28.5, "Celsius",
"2025-06-19T14:45:00")

print(f"Coordinates: {coordinates}")
print(f"Sensor Reading Metadata: {sensor_reading_metadata}")

# Accessing elements (slicing works like lists)
print(f"Latitude: {coordinates[0]}")
print(f"Sensor ID: {sensor_reading_metadata[0]}")

# Attempting to modify a tuple (will raise TypeError)
try:
    coordinates[0] = 11.0
except TypeError as e:
    print(f"Error attempting to modify tuple: {e}")

# Tuples can be "unpacked"
lat, lon, alt = coordinates
print(f"Unpacked: Lat={lat}, Lon={lon}, Alt={alt}")

# A single-element tuple needs a comma to differentiate from a parenthesized
expression
single_tuple = (10,)
not_a_tuple = (10)

print(f"Single tuple type: {type(single_tuple)}, Not a tuple type:
{type(not_a_tuple)}")
```


Use Cases for Tuples:

- Representing fixed collections of related data (e.g., coordinates, RGB color values, database records).
- Function arguments and return values where the order and immutability are important.
- Using dictionary keys (because they are immutable and thus hashable).

3.3 Sets

Sets are unordered collections of **unique** elements. They are defined by curly braces {} or the `set()` constructor.

Set Characteristics

Characteristic	Description
Unordered	Elements do not have a defined order; order is not preserved.
Mutable	Elements can be added or removed after creation.
Unique Elements	Automatically removes duplicate elements.
Unindexed	Elements cannot be accessed by index.

Python

```
# Creating sets
sensor_types = {"Temperature", "Pressure", "Humidity", "Temperature"} #
"Temperature" is duplicated but stored once
print(f"Sensor Types: {sensor_types}") # Order may vary

known_protocols = set(["TCP", "UDP", "HTTP", "TCP"])
print(f"Known Protocols: {known_protocols}")

# Adding and removing elements
sensor_types.add("Light")
print(f"After adding 'Light': {sensor_types}")
```

```

sensor_types.remove("Humidity")
print(f"After removing 'Humidity': {sensor_types}")

# Set operations (useful for data filtering/comparison)
active_sensors = {"Temperature", "Pressure", "Flow"}
alert_sensors = {"Pressure", "Humidity", "Voltage"}

# Union: all elements from both sets
all_sensors = active_sensors.union(alert_sensors)
print(f"All sensors (Union): {all_sensors}")

# Intersection: elements common to both sets
common_sensors = active_sensors.intersection(alert_sensors)
print(f"Common sensors (Intersection): {common_sensors}")

# Difference: elements in active_sensors but not in alert_sensors
only_active = active_sensors.difference(alert_sensors)
print(f"Only active sensors: {only_active}")

# Check for element existence
if "Flow" in active_sensors:
    print("Flow sensor is present.")

```

Use Cases for Sets:

- Storing unique identifiers (e.g., unique sensor IDs, unique error codes).
- Efficiently checking for membership (in operator).
- Performing mathematical set operations (union, intersection, difference) on collections of items.
- Removing duplicates from a list.

3.4 Dictionaries

Dictionaries are unordered collections of key-value pairs. Each key must be unique and immutable, and it maps to a value. They are defined by curly braces {} with key: value pairs.

Dictionary Characteristics

Characteristic	Description
Unordered	Elements do not have a defined order (in Python 3.7+ insertion order is preserved, but still considered logically unordered for general purpose).
Mutable	Key-value pairs can be added, removed, or updated after creation.
Key-Value Pairs	Stores data as key: value mappings.
Unique Keys	Each key must be unique within a dictionary.
Keys are Immutable	Keys must be of an immutable type (strings, numbers, tuples).

Python

```
# Creating dictionaries
# Sensor configuration dictionary
sensor_config = {
    "temp_sensor_id": "TS_001",
    "pressure_sensor_id": "PS_005",
    "location": "Engine Bay",
    "sampling_rate_hz": 10
}

# Another way to create using dict() constructor with keywords
device_status = dict(status="Operational", last_check="2025-06-19",
    uptime_hours=72.5)

print(f"Sensor Config: {sensor_config}")
print(f"Device Status: {device_status}")

# Accessing values by key
print(f"Temp Sensor ID: {sensor_config['temp_sensor_id']}")
print(f"Device Status: {device_status['status']}")

# Adding/modifying elements
sensor_config["manufacturer"] = "Acme Corp" # Add a new key-value pair
sensor_config["sampling_rate_hz"] = 20 # Update an existing value
print(f"Updated Sensor Config: {sensor_config}")
```

```

# Removing elements
del sensor_config["location"] # Delete by key
print(f"After deleting location: {sensor_config}")

# Iterating through dictionaries
print("\nIterating through sensor config:")
print("Keys:")
for key in sensor_config.keys():
    print(key)

print("\nValues:")
for value in sensor_config.values():
    print(value)

print("\nKey-Value Pairs:")
for key, value in sensor_config.items():
    print(f"{key}: {value}")

# Check for key existence
if "sampling_rate_hz" in sensor_config:
    print("Sampling rate exists in config.")

# Using get() method for safe access (returns None or default if key not found)
non_existent = sensor_config.get("calibration_date")
print(f"Calibration Date (using get): {non_existent}")
calibration_date = sensor_config.get("calibration_date", "N/A")
print(f"Calibration Date (using get with default): {calibration_date}")

```

Use Cases for Dictionaries:

- Storing structured data where elements are accessed by meaningful keys (e.g., configuration settings, patient records, metadata).
- Representing JSON-like data structures (common in APIs).
- Implementing lookup tables or mappings.
- Counting occurrences of items.

4. Comprehensions with Conditions

Comprehensions (List, Dictionary, and Set) provide a concise and efficient way to create new sequences or collections from existing ones. They are highly "Pythonic" and often more readable and faster than traditional for loops, especially when including conditional logic.

4.1 List Comprehensions

Syntax: [expression for item in iterable if condition]

Python

```
# Example 1: Filtering sensor data to include only readings above a threshold
raw_readings_mV = [25, 120, 155, 80, 220]
alert_threshold_mV = 150

high_readings = [reading for reading in raw_readings_mV if reading >
alert_threshold_mV]
print(f"High Readings (mV): {high_readings}")
```

Python

```
# Example 2: Normalizing and converting units for selected readings
# Convert mV to Volts (divide by 1000) and only include readings above 100 mV
normalized_volts = [
    reading / 1000
    for reading in raw_readings_mV
    if reading > 100
]
print(f"Normalized Readings (V, >100mV): {normalized_volts}")
```

Python

```
# Example 3: Nested list comprehension (e.g., processing a 2D grid)
matrix = [
    [1,2,3],
    [4,5,6],
    [7,8,9]
]

# Flatten the matrix and square only even numbers
processed_elements = [
    element**2
    for row in matrix
    for element in row
    if element % 2 == 0
]
print(f"Processed elements (squared evens): {processed_elements}")
```

4.2 Dictionary Comprehensions

Syntax: {key_expression: value_expression for item in iterable if condition}

Python

```
# Example 1: Creating a dictionary from a list of sensor IDs with default status
sensor_ids = ["TS_001", "PS_002", "HS_003", "VS_004"]

sensor_status = {sensor_id: "Operational" for sensor_id in sensor_ids}
print(f"Sensor Status: {sensor_status}")

sensor_status = {sensor_id: "Operational" for sensor_id in sensor_ids}
print(f"Sensor Status: {sensor_status}")
```

Python

```
# Example 2: Filtering and transforming existing dictionary data
# Create a new dictionary for critical parameters (values > 100)
system_parameters = {
```

```

    "cpu_temp": 85,
    "memory_usage_mb": 2048,
    "disk_free_gb": 150,
    "network_latency_ms": 120,
    "power_draw_watts": 350
}

critical_alerts = {
    param: value
    for param, value in system_parameters.items()
    if value > 100 # Arbitrary threshold for critical alerts
}
print(f"Critical Alerts: {critical_alerts}")

```

Python

```

# Example 3: Mapping string data to a numerical ID if it starts with 'TS'
device_codes = ["TS_001", "PS_002", "TS_003", "HS_004"]
device_numerical_map = {
    code: int(code.split('_')[1])
    for code in device_codes
    if code.startswith('TS')
}
print(f"Temperature Sensor Numerical Map: {device_numerical_map}")

```

4.3 Set Comprehensions

Syntax: {expression for item in iterable if condition} (similar to list comprehension but produces unique elements)

Python

```

# Example 1: Extracting unique error codes from a log list
all_error_codes = [
    "E101", "W203", "E101", "F001", "I500", "W203", "E102", "E101"
]

```

```
unique_error_ids = {code for code in all_error_codes if code.startswith('E')}  
print(f"Unique Error IDs: {unique_error_ids}")
```

Python

```
# Example 2: Generating a set of squared odd numbers from a range  
squared_odd_numbers = {x**2 for x in range(1, 11) if x % 2 != 0}  
print(f"Squared Odd Numbers (1-10): {squared_odd_numbers}")
```

Benefits of Comprehensions:

- **Conciseness:** Reduces multiple lines of code into a single, compact expression.
- **Readability:** Once understood, they are often easier to read than explicit for loops for simple transformations/filters.
- **Performance:** Generally faster than equivalent for loops due to internal optimizations.

5. Practical Lab: Data Normalization and Encoding

This section outlines concepts for a hands-on lab exercise. Data normalization and encoding are critical preprocessing steps in scientific data analysis, especially before applying machine learning algorithms or comparing disparate datasets.

5.1 Data Normalization Concepts

Purpose: To scale numerical data into a "normal" range (e.g., 0-1 or -1 to 1) to prevent features with larger numerical ranges from dominating those with smaller ranges, and to improve algorithm performance.

Common Techniques:

- **Min-Max Scaling (Normalization):** Scales features to a fixed range, usually 0 to 1.
 - Formula: $X_{normalized} = (X - X_{min}) / (X_{max} - X_{min})$
- **Standardization (Z-score Normalization):** Scales features to have a mean of 0 and a standard deviation of 1.
 - Formula: $X_{standardized} = (X - \mu) / \sigma$ (where μ is mean, σ is standard deviation)

Lab Tasks (Conceptual):

1. **Scenario:** You have sensor readings (e.g., temperature in Celsius, pressure in kPa) with different ranges.
2. **Task 1: Min-Max Scale Temperature Data:**
 - Generate a list of hypothetical temperature readings.
 - Find the minimum and maximum temperature values.
 - Apply the Min-Max scaling formula to normalize these readings to the 0-1 range.

- Store results in a new list.

3. Task 2: Standardize Pressure Data:

- Generate a list of hypothetical pressure readings.
- Calculate the mean and standard deviation of these readings (using `sum()` and `len()` for mean, and a loop for std dev, or hint at `numpy.mean`, `numpy.std` for later).
- Apply the standardization formula.
- Store results in a new list.

5.2 Data Encoding Concepts

Purpose: To convert categorical (non-numerical) data into a numerical format that can be used by algorithms.

Common Techniques:

- **One-Hot Encoding:** Converts categorical variables into a binary (0 or 1) numerical format. Each category value is converted into a new column, and assigned a 1 or 0 value.
 - Example: If you have a 'Sensor Type' column with 'Temperature', 'Pressure', 'Humidity', it becomes three new columns: 'Sensor_Temperature', 'Sensor_Pressure', 'Sensor_Humidity'.
- **Label Encoding:** Assigns a unique integer to each category.
 - Example: 'Temperature' -> 0, 'Pressure' -> 1, 'Humidity' -> 2.

Lab Tasks (Conceptual):

1. **Scenario:** You have a list of sensor states (e.g., "Active", "Standby", "Error", "Active").
2. **Task 1: Implement One-Hot Encoding:**

- Identify all unique sensor states.
- For each sensor state in the original list, create a corresponding "one-hot" representation (e.g., a list of 0s and 1s, or a dictionary).
- Consider how this would extend to a tabular dataset (e.g., using a dictionary to map states to their one-hot vectors).

3. Task 2: Implement Label Encoding:

- Create a mapping (e.g., a dictionary) from each unique sensor state to a unique integer.
- Apply this mapping to the original list of sensor states to create a new list of numerical labels.

Python

```
# Conceptual code for Lab (for illustration - actual implementation is part of the lab)
```

```
# Data Normalization
```

```
# raw_temp = [20.0, 22.5, 21.0, 28.0, 19.5]
```

```
# min_temp = min(raw_temp)
```

```
# max_temp = max(raw_temp)
```

```
# normalized_temp = [(t - min_temp) / (max_temp - min_temp) for t in raw_temp]
```

```
# print(f"Normalized Temperature: {normalized_temp}")
```

```
# Data Encoding
```

```
# sensor_states = ["Active", "Standby", "Error", "Active", "Standby"]
```

```
# unique_states = sorted(list(set(sensor_states)))
```

```
# print(f"Unique States: {unique_states}")
```

```
# One-Hot Encoding (conceptual)
```

```
# one_hot_map = {state: [1 if state == u_state else 0 for u_state in unique_states] for state in unique_states}
```

```
# print(f"One-Hot Map: {one_hot_map}")
```

```
# encoded_states_one_hot = [one_hot_map[state] for state in sensor_states]
```

```
# print(f"One-Hot Encoded States: {encoded_states_one_hot}")
```

```
# Label Encoding (conceptual)
```

```
# label_map = {state: i for i, state in enumerate(unique_states)}
```

```
# print(f"Label Map: {label_map}")
```

```
# encoded_states_label = [label_map[state] for state in sensor_states]
```

```
# print(f"Label Encoded States: {encoded_states_label}")
```

Python

```
# Data Normalization
raw_temp = [20.0, 22.5, 21.0, 28.0, 19.5]
min_temp = min(raw_temp)
max_temp = max(raw_temp)
normalized_temp = [(t - min_temp) / (max_temp - min_temp) for t in raw_temp]
print(f"Normalized Temperature: {normalized_temp}")

print("-" * 60)

# Data Encoding
sensor_states = ["Active", "Standby", "Error", "Active", "Standby"]
unique_states = sorted(list(set(sensor_states)))
print(f"Unique States: {unique_states}")




# One-Hot Encoding
one_hot_map = {
    state: [1 if state == u_state else 0 for u_state in unique_states]
    for state in unique_states
}
print(f"One-Hot Map: {one_hot_map}")
encoded_states_one_hot = [one_hot_map[state] for state in sensor_states]
print(f"One-Hot Encoded States: {encoded_states_one_hot}")

print("-" * 60)

# Label Encoding
label_map = {state: i for i, state in enumerate(unique_states)}
print(f"Label Map: {label_map}")
encoded_states_label = [label_map[state] for state in sensor_states]
print(f"Label Encoded States: {encoded_states_label}")
```

This notebook provides a foundational understanding of Python's core data types and operations, along with practical insights into data preprocessing techniques. Mastering these concepts is essential for manipulating and preparing the diverse datasets encountered in scientific and engineering projects.

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.
