

Day 1 - Session 1:

Python Overview & Coding Practices - Deep Dive

computing, an overview of the Spyder IDE, core Python syntax principles (PEP 8, dynamic typing), and fundamental concepts like mutable vs. immutable types and type hinting. Each section aims for a deeper understanding, crucial for robust engineering and scientific applications.

1. Python Overview in Scientific Computing

- **Real-World Applications (ISRO/URSC Context):**
 - **Data Acquisition & Processing:** Interfacing with instruments (e.g., using pyVISA, pySerial for lab equipment, as will be discussed on Day 5) to collect, process, and analyze sensor data from satellites, ground stations, or test setups.
 - **Telemetry Analysis:** Parsing, filtering, visualizing, and performing statistical analysis on telemetry streams from spacecraft.
 - **Scientific Modeling & Simulation:** Developing numerical models for orbital mechanics, atmospheric phenomena, material science, or signal propagation.

Python's rise in scientific and engineering domains isn't accidental. Its high-level nature, combined with a powerful ecosystem of specialized libraries, makes it an ideal tool for rapid prototyping, complex data analysis, simulation, and automation.

1.1 Why Python for Scientific & Engineering Applications?

- **High-Level & Interpreted:** Python allows for significantly faster development cycles compared to lower-level, compiled languages like C++ or Fortran. This rapid iteration is invaluable in research and development where algorithms are constantly being refined. While execution speed can be a concern, critical performance bottlenecks are often handled by underlying libraries written in C/Fortran.
- **Extensive Ecosystem of Libraries:**
 - **NumPy:** The fundamental package for numerical computation, providing high-performance multidimensional array objects and tools for working with these arrays. Essential for linear algebra, Fourier transforms, etc.
 - **SciPy:** Builds on NumPy, offering modules for optimization, linear algebra,

- **Automation:** Scripting repetitive tasks in data workflows, test benches, or mission control systems.
- **Image Processing:** Analyzing satellite imagery or experimental visual data.

1.2 Python's Simplicity vs. Verbosity (Illustrative)

Let's look at a trivial example to illustrate Python's conciseness for mathematical operations compared to a more verbose language.

C++ (Conceptual)

```
C/C++
// Imagine this C++ Code

#include <iostream>

int main() {
    double radius = 5.0;
    const double PI = 3.14159;
    double area = PI * radius * radius;
    std::cout << "Area: " << area << std::endl;
    return 0;
}
```

Python (Equivalent)

```
Python
# Python code for the same calculation
radius = 5.0
PI = 3.14159
area = PI * radius * radius
print(f"Area: {area}")
```

The Python code is noticeably shorter and often more intuitive to read, which speeds up development and debugging, especially for complex scientific algorithms.

1.3 Hinting at Future Capabilities

Here, we just import NumPy to give a glimpse of the powerful libraries that will be explored in depth later.

```
Python
import numpy as np

# Create a NumPy array
data = np.array([10.5, 20.3, 15.7, 22.1, 18.9])
print(f"Our 'sensor data' array: {data}")
print(f"Mean of data: {np.mean(data)}")
```

2. Python Environments: Managing Dependencies

As Python projects grow in complexity, managing their dependencies becomes crucial. Different projects might require different versions of the same library, or even different Python versions. This is where Python environments come in. An environment is an isolated space where you can install specific versions of Python and libraries without conflicting with other projects on your system.

2.1 Why Use Environments?

- Isolation: Prevent conflicts between project dependencies. Project A needs `requests==2.20` while Project B needs `requests==2.28`. Environments allow both to coexist.
- Reproducibility: Easily share your project with others, ensuring they can install the exact same dependencies and run your code correctly.
- Cleanliness: Keep your global Python installation clean, avoiding a cluttered mess of packages.
- Specific Python Versions: Work with different Python versions for different projects (e.g., Python 3.8 for an older project, Python 3.10 for a new one).

2.2 Types of Python Environments

The two most common types of environments you'll encounter are `venv` (Virtual Environments) and `conda` environments.

2.2.1 Virtual Environments (venv)

`venv` is a module built into Python's standard library (since Python 3.3). It's the standard way to create lightweight, isolated Python environments.

- Purpose: Primarily designed for managing Python package dependencies. It copies a minimal Python interpreter and then installs packages via `pip` into that isolated directory.
- Pros:
 - Lightweight: Part of the standard library, no external tools needed.
 - Simplicity: Easy to create and use for Python-only projects.
 - Universally Supported: Works across most operating systems.
- Cons:
 - Python-centric: Only manages Python packages. It doesn't help with non-Python dependencies (e.g., C/C++ libraries that scientific packages often rely on).
 - No Python Version Management: You can only create a `venv` for the Python version you use to create it. To use a different Python version, you need that version installed globally first.

How to Use `venv` (Terminal Commands):

1. Create an environment:

Shell

```
python -m venv my_project_env
```

2. This creates a directory named `my_project_env` containing the environment.
3. Activate the environment:

- On Windows

Shell

```
.\my_project_env\Scripts\activate
```

- On macOS/Linux:

Shell

```
source my_project_env/bin/activate
```

4. Once activated, your terminal prompt will typically show the environment's name (e.g., `(my_project_env) your_username@your_machine:~$`). All `pip install` commands will now install packages into *this* environment.

5. Install packages:

Shell

```
pip install pandas matplotlib
```

6. Deactivate the environment:

Shell

```
deactivate
```

2.2.2 Conda Environments (Anaconda/Miniconda)

Conda is a cross-platform package and environment manager that comes with Anaconda (a distribution for scientific computing) or Miniconda (a minimal installer for conda). It's more powerful than `venv` because it can manage:

- Python versions: You can create environments with specific Python versions (e.g., Python 3.9, 3.10, 3.11).
- Non-Python packages: Crucially, it can install and manage system-level libraries, compilers, and other software that many scientific Python packages (like NumPy, SciPy, TensorFlow) depend on. This often resolves complex installation issues.
- Language-agnostic: While popular for Python, Conda can manage environments for R, Java, and other languages.

- Pros:
 - Comprehensive: Manages Python versions and non-Python dependencies.
 - Robust for Scientific Stacks: Excellent for complex scientific libraries that have external dependencies.
 - Cross-platform: Works seamlessly on Windows, macOS, and Linux.
- Cons:
 - Larger Footprint: Conda environments can be larger than `venv` environments due to managing more binaries.
 - Requires Installation: You need to install Anaconda or Miniconda first.

How to Use `conda` (Terminal Commands):

1. Create an environment (with a specific Python version and packages):

Shell

```
conda create --name my_science_env python=3.9 numpy pandas
```

2. (You can omit `python=3.9` to use the default Python version if you prefer).

3. Activate the environment:

Shell

```
conda activate my_science_env
```

4. Your prompt will change to `(my_science_env)`.

5. Install more packages:

Shell

```
conda install scipy matplotlib  
# Or, if a package is not available on conda-forge:
```

```
# pip install my_custom_package
```

6. Deactivate the environment:

Shell

```
conda deactivate
```

2.3 Choosing Between **venv** and **conda**

- Use **venv** if:
 - Your project is primarily Python-only and doesn't have complex non-Python dependencies.
 - You want a lightweight solution without installing Anaconda/Miniconda.
 - You are comfortable managing Python versions separately.
- Use **conda** if:
 - You are working in scientific computing, data science, or machine learning, where projects often rely on C/Fortran libraries.
 - You need to manage different Python versions for different projects.
 - You prefer a single tool to manage all your package dependencies (both Python and non-Python).
 - You are already using Anaconda/Miniconda for other reasons (e.g., Spyder, Jupyter Lab).

3. Spyder IDE: Environment, Panels, Console

Spyder (Scientific PYthon Development EnviRonment) is a powerful open-source IDE optimized for scientific workflows, often included with the Anaconda distribution. It provides an integrated experience for coding, debugging, and data exploration.

3.1 Key Panels and Their Functionality

While we can't interact with Spyder directly in this notebook, understanding its layout is crucial for efficient development.

- **Editor:** (Top-left usually) This is where you write and edit your Python scripts (.py files). It offers features like:
 - Syntax highlighting (colors for keywords, strings, etc.)
 - Code completion (suggests variable names, functions as you type)
 - Code analysis (identifies potential errors or style violations)
 - Integrated debugger for setting breakpoints and stepping through code.
- **IPython Console:** (Bottom-right usually) An interactive shell where you can execute Python code line by line or run entire scripts. It's excellent for:
 - Testing small snippets of code immediately.
 - Debugging by inspecting variable states at runtime.
 - Exploring modules and functions.
 - The In [X]: and Out [X]: prompts make it easy to track execution history.
- **Variable Explorer:** (Top-right usually) This is a standout feature for scientific users. It displays all variables currently defined in your IPython console's

namespace, allowing you to:

- Inspect their names, types, sizes/shapes (especially useful for NumPy arrays and Pandas DataFrames), and values.
- Double-click to open data structures like DataFrames, lists, or dictionaries in a dedicated viewer, similar to a spreadsheet. This is invaluable for debugging data transformations.
- **Plots Pane:** (Often grouped with Variable Explorer) Displays all plots generated by libraries like Matplotlib, allowing you to view and save them easily without external pop-up windows.
- **File Explorer:** (Often grouped with Variable Explorer) A basic file browser to navigate your project directory.
- **Help Pane:** (Often grouped with Variable Explorer) Provides quick access to documentation for functions, modules, and classes. Just type `?function_name` in the console or press `Ctrl+I` (or `Cmd+I` on Mac) on a function name in the editor.

3.2 Environment Management (within Anaconda/Spyder)

Anaconda simplifies environment management. An "environment" is an isolated space where you can install specific versions of Python and libraries without conflicting with other projects.

- **Why environments?** Imagine Project A needs numpy version 1.20 and Project B needs numpy version 1.25. Without environments, installing one might break the other. Environments solve this.
- **Managing in Spyder:** Spyder typically has a dropdown or setting (often under Tools -> Preferences -> Python interpreter) where you can select which Anaconda environment your current session should use.

3.3 Other Popular IDEs and Development Environments

While Spyder is excellent for scientific computing, many other powerful IDEs and environments cater to different preferences and project types.

- **Jupyter Notebook / JupyterLab:**
 - **Description:** An interactive web-based environment that allows you to create and share documents containing live code, equations,

visualizations, and narrative text. It's widely used in data science, machine learning, and scientific research for its ability to combine code execution with rich documentation. JupyterLab is the next-generation interface for Jupyter, offering a more flexible and integrated development experience.

- Pros: Excellent for exploratory data analysis, prototyping, sharing reproducible research, and teaching. Supports many languages (kernels).
- Cons: Not a full-fledged IDE for large-scale software development; limited debugging features compared to traditional IDEs.
- Best For: Data exploration, quick scripts, interactive reports, presentations, and teaching.

- **VS Code (Visual Studio Code):**

- Description: A lightweight, powerful, and highly customizable code editor developed by Microsoft. It supports Python development through a rich ecosystem of extensions, offering features like intelligent code completion, linting, debugging, and seamless integration with Git.
- Pros: Extremely popular, fast, extensive extensions marketplace, strong debugging capabilities, integrated terminal, excellent Git integration.
- Cons: Can require more setup (installing extensions) to get a full Python development environment compared to an out-of-the-box solution like Anaconda/Spyder.
- Best For: General-purpose Python development, web development (with Python backends), scripting, and larger software projects.

- **Google Colaboratory (Colab):**

- Description: A free cloud-based Jupyter Notebook environment that runs entirely in your browser. It provides access to powerful hardware like GPUs and TPUs, making it ideal for deep learning and heavy computational tasks without local setup.
- Pros: No setup required, free access to GPUs/TPUs, easy sharing, integrated with Google Drive.
- Cons: Sessions can time out, require an internet connection, not suitable for persistent local development.
- Best For: Deep learning experimentation, collaborative research, quick prototyping of machine learning models, and educational purposes.

- **PyCharm (Community/Professional):**

- Description: A dedicated Python IDE developed by JetBrains, known for

its powerful features, intelligent code analysis, and extensive refactoring capabilities. The Community Edition is free and open-source.

- Pros: Best-in-class code analysis, refactoring tools, powerful debugger, excellent support for frameworks (Django, Flask), database tools (Professional).
- Cons: Can be resource-intensive, steeper learning curve for beginners compared to lighter editors.
- Best For: Professional Python development, large-scale projects, enterprise applications, and web development.

- **IDLE:**

- Description: Python's default Integrated Development and Learning Environment. It's a simple IDE that comes bundled with Python itself.
- Pros: Always available with Python installation, simple for basic scripting and learning.
- Cons: Very basic features, not suitable for complex projects or scientific computing.
- Best For: Absolute beginners, very small scripts, and learning Python fundamentals.

The choice of IDE often depends on the specific project requirements, team preferences, and individual comfort levels. For scientific and engineering tasks, Jupyter Notebooks (especially with JupyterLab or Colab) and VS Code are increasingly popular alongside traditional IDEs like Spyder and PyCharm.

3.4 Summary of Development Environments

Environment Type	Key Features	Best For	Typical Use Case (ISRO/URSC Context)
Traditional IDEs (Spyder, PyCharm)	Integrated debugging, advanced code analysis, refactoring tools, project management.	Large-scale development, complex applications, professional software engineering.	Developing robust software for data acquisition systems, control systems, or complex simulation frameworks.

Notebook Environments (Jupyter, Colab)	Interactive code execution, inline visualizations, mixed media (text, code, plots), easy sharing.	Exploratory data analysis, rapid prototyping, research, teaching, collaborative work.	Analyzing telemetry data, visualizing sensor outputs, developing and sharing scientific algorithms.
Lightweight Editors (VS Code)	Fast, highly customizable via extensions, good balance of features for various project sizes.	General-purpose scripting, web development, moderate-sized projects, flexible development.	Scripting automation tasks, developing smaller utilities, contributing to open-source projects.
Basic Editors (IDLE)	Minimalist, simple, always available.	Absolute beginners, very simple scripts, quick testing of syntax.	Learning Python basics, immediate short code tests.

4. Python Syntax Recap (PEP8, Dynamic Typing)

A strong grasp of Python's fundamental syntax and best practices is essential for writing maintainable, readable, and collaborative code.

4.1 PEP 8 - The Style Guide for Python Code

PEP 8 (Python Enhancement Proposal 8) provides conventions for writing Python code. Adhering to it makes your code consistent with the broader Python community, enhancing readability and collaboration. It's not mandatory, but highly recommended.

Why PEP 8?

- **Readability:** Consistent formatting makes code easier to read and understand, for yourself and others.
- **Maintainability:** Easier to debug and update code written in a consistent style.
- **Collaboration:** Essential for teams working on the same codebase, ensuring a uniform appearance.

Key PEP 8 Guidelines (and Examples):

- **Indentation:** Use 4 spaces per indentation level. Never use tabs.

Python

```
# Good
def calculate_area(radius):
    if radius > 0:
        area = 3.14159 * radius ** 2
        return area
    else:
        return 0

# Bad (incorrect indentation)
# def calculate_area(radius):
#     if radius > 0:
#         area = 3.14159 * radius ** 2 # Incorrect indentation here
#         return area
#     else:
#         return 0
```

- Line Length: Limit all lines to a maximum of 79 characters. For longer lines, use parentheses, brackets, or backslashes for implicit or explicit line continuation.

Python

```
# Good (implicit line continuation)
long_list_of_values = [
    1.2345, 2.3456, 3.4567, 4.5678, 5.6789,
    6.7890, 7.8901, 8.9012, 9.0123, 10.1234
]

# Good (explicit line continuation - less common for lists, but possible)
# very_long_variable_name_to_demonstrate_line_break = \
#     another_very_long_variable_name + some_other_long_expression
```

- Naming Conventions:
 - `snake_case` for variables, functions, and methods (`calculate_power_output`).
 - `PascalCase` for classes (`SensorDataProcessor`).
 - `ALL_CAPS` for constants (`PI_VALUE = 3.14159`).
 - Leading underscore “_” for internal-use only (non-public) methods/attributes (`_internal_helper`).
 - Double leading/trailing `underscores` `__` for special methods (dunder methods, e.g., `__init__`).

Python

```
# Good naming conventions
class TelemetryPacket:
    MAX_SIGNAL_STRENGTH = 100.0

    def __init__(self, timestamp, sensor_reading):
        self.timestamp = timestamp
        self.sensor_reading = sensor_reading

    def process_data(self):
        # ... processing logic ...
        return self.sensor_reading * 2

# Bad naming conventions (examples)
# class telemetrypacket: # should be PascalCase
#     maxsignalstrength = 100.0 # should be ALL_CAPS
#     def processData(self): # should be snake_case
#         pass
```

- **Blank Lines:** Use two blank lines to separate top-level function and class definitions. Use single blank lines to separate methods within a class, or logical sections within a function.

Python

```
def function_one():
    pass

def function_two():
    pass

class MyClass:
    def method_a(self):
        # Logical block 1
        x = 10
        y = 20

        # Logical block 2
        result = x + y
        print(result)

    def method_b(self):
        pass
```

- **Imports:** Place imports at the top of the file, one import per line. Group

standard library imports, then third-party imports, then local application/library imports, each group separated by a blank line.

```
Python
import os
import sys

import numpy as np
import pandas as pd

from my_module import some_function
from .subpackage import another_utility
```

Tools for PEP 8 Compliance:

IDEs like Spyder often have built-in linters (e.g., PyLint, Flake8) that can automatically check your code against PEP 8 guidelines and highlight violations.

4.2 Dynamic Typing

Python is a dynamically typed language. This means:

- You don't need to declare the type of a variable before assigning a value to it.
- The type of a variable is determined at runtime based on the value it holds.
- A variable can even hold values of different types throughout its lifetime.

Pros of Dynamic Typing:

- **Flexibility:** Faster development, as you don't need to worry about explicit type declarations.
- **Readability (for simple cases):** Code can often look cleaner and less verbose.

Cons of Dynamic Typing:

- **Runtime Errors:** Type-related errors might only appear when the code is executed, making debugging harder, especially in large applications.
- **Less Clarity:** Without type declarations, it can be less clear what type of data a function expects or returns, impacting maintainability and collaboration.

Example of Dynamic Typing:

Python

```
# 'x' initially holds an integer
x = 10
print(f"Value of x: {x}, Type of x: {type(x)}")

# Now 'x' holds a string
x = "Hello, Python\!"
print(f"Value of x: {x}, Type of x: {type(x)}")

# Now 'x' holds a float
x = 3.14159
print(f"Value of x: {x}, Type of x: {type(x)}")

# Example of a potential runtime error due to unexpected type
def add_two_numbers(a, b):
    return a + b

result1 = add_two_numbers(5, 3)
print(f"Result 1 (integers): {result1}")

# This will work because Python knows how to add strings
result2 = add_two_numbers("Hello ", "World")
print(f"Result 2 (strings): {result2}")

# This will cause a TypeError at runtime
try:
    result3 = add_two_numbers(5, "three")
    print(f"Result 3: {result3}")
except TypeError as e:
    print(f"Error: {e} - This happened because of dynamic typing allowing mixed types.")
```


5. Mutable vs. Immutable Types

Understanding whether an object is mutable or immutable is fundamental to Python programming. It dictates how objects behave when modified, copied, or passed to functions, and has significant implications for data integrity and memory management.

5.1 What's the Difference?

- **Immutable Types:** The *value* of the object cannot be changed after it is created. If you perform an operation that seems to "change" an immutable object, you are actually creating a *new* object in memory.
 - **Examples:** Numbers (int, float, complex), strings (str), tuples (tuple), frozen sets (frozenset).
- **Mutable Types:** The *value* of the object *can* be changed after it is created. Modifications happen "in-place," meaning the object itself is altered, and its memory address remains the same.
 - **Examples:** Lists (list), dictionaries (dict), sets (set), byte arrays (bytearray).

5.2 Demonstrating with id()

The `id()` function in Python returns the "identity" of an object, which is guaranteed to be unique and constant for that object during its lifetime. For CPython, this is typically the memory address of the object.

5.2.1 Immutable Examples

```
Python
# --- Integers ---
a = 10
print(f"Initial integer a: {a}, ID: {id(a)}")
b = a # 'b' now refers to the same object as 'a'
print(f"b = a, Integer b: {b}, ID: {id(b)}")

a += 1 # This creates a new integer object for 'a'
print(f"After a += 1, Integer a: {a}, ID: {id(a)}")
print(f"Integer b (unchanged): {b}, ID: {id(b)}") # b still refers to the
old 10 object

print("\n--- Strings ---")
s1 = "Hello"
print(f"Initial string s1: {s1}, ID: {id(s1)}")

s2 = s1 # s2 points to the same object
print(f"s2 = s1, String s2: {s2}, ID: {id(s2)}")

s1 += " World" # This creates a *new* string object for s1
print(f"After s1 += ' World', String s1: {s1}, ID: {id(s1)}")
print(f"String s2 (unchanged): {s2}, ID: {id(s2)}") # s2 still points to the
original "Hello"

print("\n--- Tuples ---")
t1 = (1, 2, 3)
print(f"Initial tuple t1: {t1}, ID: {id(t1)}")

# Attempting to modify a tuple raises an error
try:
    t1 = 99
except TypeError as e:
    print(f"Error attempting to modify tuple: {e}")

# If we 'reassign' a tuple, it's a new tuple
t1 = (4, 5, 6)
print(f"After reassigning t1: {t1}, ID: {id(t1)}")
```

5.2.2 Mutable Examples

```
Python
print("--- Lists ---")
list1 = [1, 2, 3, True, 'ABC']
```

```

print(f"Initial list1: {list1}, ID: {id(list1)}")

list2 = list1 # list2 refers to the *same* object as list1
print(f"list2 = list1, List2: {list2}, ID: {id(list2)}")

list1.append(4) # In-place modification
print(f"After list1.append(4), List1: {list1}, ID: {id(list1)}")
print(f"List2 (changed\!): {list2}, ID: {id(list2)}") # list2 reflects the
change\!

print("\n--- Dictionaries ---")
dict1 = {'a': 1, 'b': 2}
print(f"Initial dict1: {dict1}, ID: {id(dict1)}")

dict2 = dict1 # dict2 refers to the *same* object as dict1
print(f"dict2 = dict1, Dict2: {dict2}, ID: {id(dict2)}")

dict1['c'] = 3 # In-place modification
print(f"After dict1['c'] = 3, Dict1: {dict1}, ID: {id(dict1)}")
print(f"Dict2 (changed\!): {dict2}, ID: {id(dict2)}") # dict2 reflects the
change\!

print("\n--- Sets ---")
set1 = {1, 2, 3}
print(f"Initial set1: {set1}, ID: {id(set1)}")

set2 = set1
print(f"set2 = set1, Set2: {set2}, ID: {id(set2)}")

set1.add(4) # In-place modification
print(f"After set1.add(4), Set1: {set1}, ID: {id(set1)}")
print(f"Set2 (changed\!): {set2}, ID: {id(set2)}")

```

5.3 Practical Implications

- **Unexpected Side Effects:** Modifying a mutable object that is referenced by multiple variables (or passed to multiple functions) can lead to unexpected changes in other parts of your code. This is a common source of bugs.
- **Function Arguments:**
 - If you pass an **immutable** object to a function, any modification *within the function* will create a *new local object*, leaving the original outside the function untouched.
 - If you pass a **mutable** object to a function, the function can *directly modify the original object*.

6. Type Hints (typing module) and Annotations

Building upon the understanding of dynamic typing, **type hints** allow you to add optional type declarations to your Python code. While Python remains dynamically typed at runtime, these hints are invaluable for static analysis tools (like linters and IDEs) and for improving code readability and maintainability.

6.1 The Need for Type Hints

In large codebases, especially in scientific and engineering projects where data types are critical, dynamic typing can make it hard to infer what types functions expect or return.

Consider: `def process_data(data): ...`

Without hints, `data` could be a list of floats, a NumPy array, a Pandas DataFrame, or even a string. This ambiguity can lead to errors and confusion.

6.2 Introduction to Type Hints (PEP 484)

Type hints were introduced in PEP 484. They are *not* enforced by the Python interpreter at runtime (they are "hints"). Their primary benefits are:

- **Improved Readability:** Code becomes self-documenting regarding expected data types.
- **Enhanced Tooling Support:**
 - **IDEs (like Spyder):** Provide better autocompletion, refactoring tools, and

static error checking (e.g., flagging when you pass an int to a function expecting a str).

- **Linters/Static Analyzers (e.g., MyPy):** Can catch type-related bugs *before* your code runs.
- **Better Maintainability:** Easier to understand and modify code written by others or by yourself in the past.
- **Facilitates Collaboration:** Reduces miscommunications about data structures within a team.

6.3 Common Types from the typing Module

The built-in types (like int, str, list) can be used directly. For more complex types, especially collections and unions, you'll use the typing module.

Python

```
from typing import List, Dict, Tuple, Set, Union, Optional, Any, Callable
```

Basic Types

```
def greet(name: str) -> str:
    return f"Hello, {name}!"
```

```
print(greet("Engineers"))
```

```
# print(greet(123)) # MyPy/IDE would flag this as a type error
```

List of a specific type

```
def calculate_average(numbers: List[float]) -> float:
    if not numbers:
        return 0.0
    return sum(numbers) / len(numbers)
```

```
sensor_readings: List[float] = [12.5, 13.1, 11.9, 14.0]
```

```
print(f"Average sensor reading: {calculate_average(sensor_readings)}")
```

Python

Dictionary with specific key and value types

```
def process_config(config: Dict[str, Union[str, int]]) -> Dict[str, str]:
    processed = {}
    for key, value in config.items():
        processed[key] = str(value).upper()
```

```

        return processed

system_config: Dict[str, Union[str, int]] = {"mode": "auto", "threshold":
50, "unit": "kPa"}
print(f"Processed config: {process_config(system_config)}")

```

Python

```

# Tuple with specific types
def get_coordinates() -> Tuple[float, float, float]:
    return (10.5, 20.3, 5.1)

x, y, z = get_coordinates()
print(f"Coordinates: X={x}, Y={y}, Z={z}")

# Set of a specific type
def unique_ids(ids: Set[int]) -> int:
    return len(ids)

device_ids: Set[int] = {101, 105, 101, 109} # Duplicates are automatically
handled by set
print(f"Number of unique device IDs: {unique_ids(device_ids)}")

```

Python

```

# Union: accepts one of several types
def process_input(value: Union[int, float, str]) -> str:
    return f"Input value: {value}, type: {type(value).__name__}"

print(process_input(10))
print(process_input(3.14))
print(process_input("data_string"))

# Optional: indicates a value can be either the specified type or None
# Equivalent to Union[str, None]
def get_status_message(code: int) -> Optional[str]:
    if code == 200:
        return "Operation successful"
    elif code == 404:
        return "Resource not found"
    return None # Explicitly returns None

print(f"Status for 200: {get_status_message(200)}")
print(f"Status for 404: {get_status_message(404)}")

```

```
print(f"Status for 500: {get_status_message(500)}")
```

Python

```
# Any: use when the type is unknown or can vary widely (use sparingly)
def log_event(data: Any) -> None:
    print(f"Logging event data: {data} (Type: {type(data).__name__})")

log_event({"message": "Sensor online", "level": "INFO"})
log_event(12345)
```

Python

```
# Callable: for functions as arguments
def apply_operation(numbers: List[float], op: Callable[[float, float],
float]) -> List[float]:
    results = []
    if len(numbers) < 2:
        return numbers
    for i in range(len(numbers) - 1):
        results.append(op(numbers[i], numbers[i+1]))
    return results

def multiply(a: float, b: float) -> float:
    return a * b

print(f"Applied multiply: {apply_operation([1.0, 2.0, 3.0, 4.0],
multiply)}")

def add(a: float, b: float) -> float:
    return a + b

print(f"Applied add: {apply_operation([1.0, 2.0, 3.0, 4.0], add)}")
```

6.4 Function Annotations

Type hints are a specific use case of function annotations (PEP 3107). Annotations are

general-purpose markers for function parameters and return values. While type hinting is the most common use, you could theoretically use them for other metadata.

```
Python
# Function with an annotation for a parameter and return value
# In this case, the annotations are type hints.
def calculate_checksum(data: bytes, algorithm: str) -> int:
    """
    Calculates a checksum for the given byte data using the specified
    algorithm.
    """
    if algorithm == "crc32":
        # Simulate CRC32 calculation
        return sum(data) % 256
    else:
        raise ValueError("Unsupported algorithm")

# Accessing annotations (for advanced introspection, not common daily use)
print(f"\nAnnotations for calculate_checksum:
{calculate_checksum.__annotations__}")

# Example usage
packet_data = b'\x01\x02\x03\x04\x05'
checksum = calculate_checksum(packet_data, "crc32")
print(f"Calculated checksum: {checksum}")
```

6.5 Using Type Hints with an IDE/Linter




To fully appreciate type hints, you need to use a tool that performs static analysis. Let's imagine how an IDE like Spyder or a linter like MyPy would react to a type mismatch:

```
Python
# Imagine this code is in your IDE/editor
# IDE/MyPy would warn you if tried:
# result = calculate_average("not numbers") # Type checker would flag this

# IDE/MyPy would also help with autocompletion based on hints:
# When you type 'numbers.' after 'numbers: List[float]',
# the IDE would suggest list methods like .append(), .pop(), etc.
```

By consistently using type hints, especially in larger, more complex scientific and engineering projects, you can significantly reduce the likelihood of subtle type-related bugs and make your codebase much more robust and understandable.

ISRO URSC – Python Training | AnalogData | Rajath Kumar

 For queries: rajath@analogdata.ai |  (+91) 96633 53992 |  <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.
