Foundations for Sustainable Research Software

Shao-Ching Huang

2025-10-09

Workshop series:

1. Foundations for sustainable research software (October 9)
2. Scaling your science with parallel computing (October 16)
3. Accelerating your code with GPUs (October 30)

Resources:

- Workshop materials [link]
- UCLA Office of Advanced Research Computing [link]
- Hoffman2 Cluster [link]

Introduction

Observation

- Python has become a dominant language for scientific computing and machine learning
- Widespread use of Notebooks has led to proliferation of “linear” scripts that are difficult to reuse
- Advanced Python features for software reuse remain underutilized in research code
- Many Python codebases are limited to single-core execution
- Multi-core CPUs are standard, and GPU resources are increasingly accessible

Software Sustainability

- Keeps research code usable and maintainable over time
- Facilitates collaboration and sharing across research teams
- Prevents code from becoming obsolete or unusable
- Enables efficient updates, bug fixes, and feature enhancements
- Ensures reproducibility and transparency in scientific workflows
- Distinguishes sustainable software from disposable one-time scripts

General Goals of This Workshop Series

- Transform ad-hoc scripts into well-structured, reusable software
- Teach best practices for writing maintainable research code
- Demonstrate how to package and distribute code effectively
- Emphasize reproducibility in computational workflows
- Provide hands-on examples for scientific computing applications
- Leverage modern hardware: multi-core CPUs and GPUs
- Empower researchers to build robust, sustainable software

Why Learn This in the LLM Era?

Even with powerful LLM-based code-generating assistants, understanding software design principles remains critical:

- You provide the human intelligence
  - LLMs need clear, structured prompts that reflect good design thinking
  - You must specify what architecture, patterns, and structure you want
  - Without understanding concepts like classes, inheritance, or packaging, you can't guide the LLM effectively
- You must read and validate generated code
  - LLMs make mistakes, introduce bugs, and sometimes generate outdated or inefficient patterns
  - You need to recognize when code follows best practices vs. anti-patterns
- You maintain and evolve the codebase
  - Generated code needs to be integrated, refactored, and maintained over time
  - You make architectural decisions that LLMs cannot: "Should this be a class or a function?" "How should components interact?"
  - Long-term project success depends on human oversight of structure and design

Transitioning from Scripts to Software

A Staged Approach (using Python as an example)

- **Refactor linear scripts into functions**
  - Each function does one thing (single responsibility principle)
  - Use descriptive, meaningful function names
  - Add clear comments and documentation
- **Group related functions into classes**
- **Organize code into modules** by topic or workflow
- **Practice is essential** — mastering this skill takes time

Bottom line: for sustainable code, organize logic into functions rather than linear scripts.

Python Functions

- Functions organize code into reusable, testable blocks
- Eliminate repetition and simplify debugging and maintenance
- Encapsulate local variables within function scope

We will explore several essential features of Python functions.

Type Hinting

Improves code readability and helps with static analysis tools.

Introduced in Python 3.5 (PEP 484) in 2015.

```
count: int = 0
name: str = "Alice"
data: list[float] = [1.0, 2.5, 3.7]

from typing import Any
mixed: list[Any] = [1, "two", 3.0, [4, 5]]

def add(x: int, y: int) -> int:
  return x + y
```

Pros and Cons of Type Hints

Pros:

- Improve readability and documentation
- Catch bugs early with static analysis
- Enable better IDE support (auto-completion, type checking)
- Facilitate refactoring in large projects

Cons:

- Increase code verbosity
- Not enforced at runtime (without extra tools)
- Require extra effort for complex types
- Third-party libraries may lack type stubs

Note: Type hints do not affect compiled Python code in libraries like Numba or JAX. These libraries ignore Python type hints and use their own mechanisms (such as decorators or explicit type annotations) for optimization and compilation. Type hints are mainly for static analysis, documentation, and editor support—not for runtime or compilation behavior in Numba/JAX.

Type Hint Examples with NumPy

```python
import numpy as np
from typing import Tuple, Optional

# Calculate mean of a 1D array
def mean(arr: np.ndarray) -> float:
  return float(np.mean(arr))

# Normalize a vector
def normalize(vec: np.ndarray) -> np.ndarray:
  return vec / np.linalg.norm(vec)

# Compute dot product of two arrays
def dot(a: np.ndarray, b: np.ndarray) -> float:
  return float(np.dot(a, b))

# Return shape and dtype of an array
def array_info(arr: np.ndarray) -> Tuple[Tuple[int, ...], str]:
  return arr.shape, str(arr.dtype)

# Optional return type: can return None if input is empty
def safe_mean(arr: np.ndarray) -> Optional[float]:
  if arr.size == 0:
    return None
  return float(np.mean(arr))

# Function accepting and returning 2D arrays
def matrix_multiply(a: np.ndarray, b: np.ndarray) -> np.ndarray:
  return np.matmul(a, b)
```

Function signature

A Python function is defined by its signature, which includes:

- The function name
- The list of arguments (parameters)
- The return type (optional, with type hints)

Positional Arguments vs Keyword Arguments

Positional arguments are specified by their position in the function call:

```python
def add(x, y):
    return x + y
add(2, 3)  # x=2, y=3
```

Keyword arguments are specified by name:

```python
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")
```

```
greet(name="Alice", greeting="Hi")
```

When calling a function, you can mix positional and keyword arguments, but positional arguments must come first in the call.

```
# Valid
greet("Alice", greeting="Hi")  # positional, then keyword

# Invalid - causes SyntaxError
greet(name="Alice", "Hi")      # keyword before positional
```

Function Argument Types

- Arguments can be any Python object: int, float, str, list, dict, custom class, etc.
- You can use type hints to specify expected types:

```
def scale(x: float, factor: float) -> float:
    return x * factor
```

Return Type

- Functions can return any Python object.
- Use type hints to specify the return type:

```
def get_name() -> str:
    return "Alice"
```

Passing values to a function

- Python uses "pass by object reference" (sometimes called "pass by assignment").
- Mutable objects (lists, dicts) can be changed inside functions.
- Immutable objects (ints, strings, tuples) cannot be changed inside functions.
- Example:

  ```
  def modify_list(lst):
      lst.append(1)
  my_list = []
  modify_list(my_list)
  print(my_list)  # [1]
  ```

Global Variables

Avoid using `global` keyword unless absolutely necessary.

- Global variables are accessible throughout the module and can lead to bugs and hard-to-maintain code
- Prefer passing variables as function arguments

- Less modular, harder to test, and risk accidental modification

---

Type Aliases for Complex Types

Type aliases make code easier to read and maintain, especially for complex or repeated type hints.

Example:

```
from typing import List, Tuple, Dict

# Define a type alias for a list of 2D coordinates
Coordinates = List[Tuple[float, float]]

# Use the alias in a function signature
def total_distance(points: Coordinates) -> float:
  # ... implementation ...
  pass

# Alias for a dictionary mapping strings to lists of integers
StrToIntList = Dict[str, List[int]]
```

Type aliases are especially useful in scientific code, where data structures can be complex and reused across many functions.

Exception handling within functions

Exception handling makes functions robust by catching and managing errors gracefully.

Key behaviors:

- When an exception is raised, control immediately **returns to the caller** (or nearest `except` block)
- Remaining lines in the function are not executed
- Unhandled exceptions abort the program and return control to the operating system
- Exception handling **keeps programs running** instead of terminating unexpectedly

Use `try`, `except`, and optionally `finally` blocks:

```
def safe_divide(x: float, y: float) -> float:
  try:
    return x / y
  except ZeroDivisionError:
    print("Error: Division by zero!")
    return float('inf')

def read_file(filename: str) -> str:
  try:
    with open(filename) as f:
      return f.read()
  except FileNotFoundError:
    print(f"File not found: {filename}")
    return ""
```

You can also raise exceptions to signal errors to the caller:

```
def get_item(lst: list, idx: int):
  if idx < 0 or idx >= len(lst):
    raise IndexError("Index out of range")
  return lst[idx]
```

Default argument values

Default argument values allow parameters to have fallback values when not provided by the caller. This makes functions more flexible and easier to use.

Example:

```
def greet(name, greeting="Hello"):
  print(f"{greeting}, {name}!")

greet("Alice")           # Output: Hello, Alice!
greet("Bob", greeting="Hi")  # Output: Hi, Bob!
```

Required parameters must come before parameters with default values in the function definition.

```
# Valid
def greet(name, greeting="Hello"):  # required first, then default
    pass

# Invalid - SyntaxError
def greet(greeting="Hello", name):  # default before required
    pass
```

Keyword-only arguments

Keyword-only arguments must be specified by name when calling the function. Define them by placing a `*` in the function signature before those parameters.

Examples:

```
def example(a, b, *, c, d=5):
  print(a, b, c, d)

example(1, 2, c=3)      # c must be specified as a keyword
example(1, 2, d=7, c=4) # both c and d must be specified as keywords

def greet(name, *, greeting):  # greeting is keyword-only
    print(f"{greeting}, {name}!")

greet("Alice", "Hello")           # ERROR! Can't pass greeting positionally
greet("Alice", greeting="Hello")  # Required - must use keyword
```

Benefits: improves code clarity and prevents argument order mistakes. All parameters after `*` must be given as keywords.

---

*args and **kwargs

- `*args`: allows a function to accept any number of positional arguments (as a tuple).
- `**kwargs`: allows a function to accept any number of keyword arguments (as a dict).

Example:

```
def demo(*args, **kwargs):
    print(args)
    print(kwargs)
demo(1, 2, 3, a=4, b=5)
# Output: (1, 2, 3) {'a': 4, 'b': 5}
```

Context managers (with statement, custom via enter/exit)

Context managers are a Python feature that help you manage resources (like files, network connections, or locks) safely and automatically. They ensure setup and cleanup code runs reliably, even if errors occur.

The most common way to use a context manager is with the `with` statement.

File Handling

You will never forget to "close" a file:

```
with open("data.txt") as f:
  data = f.read()
# File is automatically closed when the block ends
```

Connecting to a database

```
import sqlite3

# Using the built-in context manager for sqlite3
with sqlite3.connect("example.db") as conn:
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE IF NOT EXISTS users (id INTEGER, name TEXT)")
    cursor.execute("INSERT INTO users VALUES (?, ?)", (1, "Alice"))
    conn.commit()
    cursor.execute("SELECT * FROM users")
    print(cursor.fetchall())
# Connection is automatically closed when the block ends
```

Custom Context Manager

You can create your own context managers by defining `__enter__` and `__exit__` methods:

```
class Timer:
    def __enter__(self):
        import time
        self.start = time.time()
        print("Timer started")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        import time
        elapsed = time.time() - self.start
        print(f"Timer stopped. Elapsed time: {elapsed:.2f}s")
        return False  # Don't suppress exceptions

# Usage
with Timer():
    # Do some work
    sum([i**2 for i in range(1000000)])
```

- The `__enter__` method runs when entering the `with` block and can return a value.
- The `__exit__` method runs when exiting the block (even if an error occurs) and receives exception information if an error happened.

Function Decorators

- Decorators are functions that modify the behavior of other functions.
- Common uses: logging, timing, access control.

Example

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before function call")
        result = func(*args, **kwargs)
        print("After function call")
        return result
    return wrapper

@my_decorator
def greet(name):
    print(f"Hello, {name}!")
greet("Alice")
```

Example

```
import time

def timing_decorator(func):
  def wrapper(*args, **kwargs):
    start = time.time()
    result = func(*args, **kwargs)
    end = time.time()
    print(f"{func.__name__} took {end - start:.4f} seconds")
    return result
  return wrapper

@timing_decorator
def slow_function():
  time.sleep(1)
  print("Done!")

slow_function()
```

Calling Functions from Other Files

- Use `import` to access functions defined in other Python files (modules).

In `utils.py`:

```
def helper():
    print("Helping!")
```

In another file:

```
from utils import helper
helper()
```

Python Classes

The Problem with Scripts

As researchers, we often start with a script:

- Load data
- Define variables for parameters
- Run functions to get results

This works, but has serious limitations:

- Data and parameters "float" around without structure
- Easy to accidentally use wrong variables
- Simple to corrupt data in ways that break later steps

Imagine a script to analyze a simulation trajectory. You might have:

```
# A collection of disconnected variables
trajectory_file = 'sim_run_01.xtc'
topology_file = 'system.pdb'
temperature = 300.0
pressure = 1.0
results = None # Will be populated later
```

```python
def load_data(traj, top):
    # ... loads data ...
    return loaded_data

def calculate_rmsd(data, temp):
    # ... does a calculation ...
    return rmsd_values

# The script runs top-to-bottom
sim_data = load_data(trajectory_file, topology_file)
results = calculate_rmsd(sim_data, temperature)
```

This is hard to reproduce. What if you forget to pass `temperature` to the `calculate_rmsd` function? What if another part of the script accidentally changes the value of `sim_data`? The state of your analysis is fragile and spread out.

An Alternative: The "Composite Function"

A common instinct for organizing this workflow without classes is to create a single "master" function that calls the other functions in sequence.

```python
def run_full_analysis(trajectory_file, topology_file, temperature, pressure):
    """Runs the entire analysis pipeline."""
    sim_data = load_data(trajectory_file, topology_file)
    results = calculate_rmsd(sim_data, temperature)
    # ... maybe a dozen more steps ...
    return results
```

This is certainly an improvement over a loose script, but it has a critical weakness that highlights the true value of classes.

The Problem with the Composite Function: Opaque State

The composite function is a **black box**. You put parameters in, and you get a final result out. But what if you want to inspect the intermediate steps? * How do you access the `sim_data` after it's been loaded? * What if you want to run an additional, exploratory analysis on the loaded data without re-running the whole pipeline? * What if you want to change a parameter and only re-run the last step?

With a composite function, you can't. The intermediate data (`sim_data`) is created and then immediately thrown away inside the function's local scope. The entire state of the analysis is temporary and inaccessible.

The Solution: A Self-Contained `Simulation` Class

A class lets us bundle all of this into a single, coherent unit.

```python
import numpy as np

class SimulationAnalysis:
    """A class to hold and analyze a single simulation run."""
    def __init__(self, trajectory_file, topology_file, temperature, pressure):
        """The constructor: sets up the object's initial state."""
        # --- Attributes: The data of the object ---
        self.trajectory_file = trajectory_file
        self.topology_file = topology_file
        self.temperature = temperature
        self.pressure = pressure

        # Internal state, starts empty
        self.data = None
        self.rmsd_results = None

    def load_data(self):
        """A method that operates on the object's own data."""
        print(f"Loading data from {self.trajectory_file}...")
        # In a real scenario, this would use a library like MDAnalysis or mdtraj
        # self.data = ... load from self.trajectory_file ...
        self.data = np.random.rand(100, 3) # Placeholder for real data

    def calculate_rmsd(self):
        """Another method. It uses the object's internal state."""
        if self.data is None:
            raise ValueError("Data not loaded. Please run .load_data() first.")

        # Notice how it uses its own attributes, like self.temperature
        print(f"Calculating RMSD at {self.temperature} K...")
        # self.rmsd_results = ... perform calculation on self.data ...
        self.rmsd_results = np.mean(self.data, axis=1) # Placeholder calculation
```

Using the Class

Now, our analysis is an “object.” All the data and logic are neatly packaged together.

```python
# Create an instance of our analysis for a specific run
sim_run_1 = SimulationAnalysis(
    trajectory_file='sim_run_01.xtc',
    topology_file='system.pdb',
    temperature=300.0,
    pressure=1.0
)

# Call the methods in a logical order
sim_run_1.load_data()
sim_run_1.calculate_rmsd()

# The results are stored safely inside the object
print(f"First 5 RMSD values: {sim_run_1.rmsd_results[:5]}")
```

Why Classes Are Critical for Reproducible Science

Funding agencies, publishers, and the scientific community increasingly demand reproducible computational research. This means code must be clear, robust, and easy for others to run and understand.

Classes achieve this in ways that are difficult without them:

1. Encapsulation for Integrity
    - Bundles data and methods together
    - Protects analysis integrity
    - Results stored safely inside objects
    - Harder to accidentally modify
    - Object acts as a "snapshot" of specific analysis
2. Clarity and Reusability
    - Makes workflow explicit
    - Easy to create independent analysis objects
    - Multiple datasets without interference

    ```
    sim_run_2 = SimulationAnalysis('sim_run_02.xtc', 'system.pdb', 310.0, 1.0)
    sim_run_2.load_data()
    sim_run_2.calculate_rmsd()
    ```

    - Nearly impossible to manage safely with floating variables
3. A Contract for Your Science
    - `__init__` defines required parameters
    - Methods define operation sequence
    - Easier for colleagues and reviewers to verify methodology
4. Enabling Safe Refactoring
    - Refactoring: restructure code to improve design without changing behavior
    - Like revising a manuscript: improve flow and clarity, not content
    - Tangled scripts are terrifying to change
    - Classes make refactoring safe through encapsulation
    - Change internal implementation without breaking other code
    - Essential for long-term research projects

Beyond Code: Reproducing the Computing Environment

True reproducibility requires more than well-structured code. The classic "it works on my computer" syndrome occurs when environments differ.

The environment must be captured and shared at multiple levels:

- Python Dependencies
    - Recreate exact package versions
    - Tools: `pip` with `requirements.txt`, `conda`, or `uv`
- System-Level Dependencies
    - C libraries or system tools
    - Tools: `conda` for non-Python software
- Entire Operating System
    - Highest level of reproducibility
    - Tools: containers (Docker, Apptainer)
    - Package code, dependencies, and OS into portable image

Two critical steps for reproducibility: 1. Use classes to organize scientific logic 2. Place code into well-defined, reproducible environment

Bottom line: Without classes, you present a script and loose variables, asking others to trust correct usage. With classes, you present a self-contained, verifiable scientific instrument.

Python Classes: Syntax and Grammar

This guide covers fundamental syntax for creating and using classes in Python.

The Core Concept: Blueprints and Instances

Two core concepts:

- Class: a blueprint defining structure and behavior
  - Example: `Molecule` class defines that all molecules have `name` and `coordinates`
- Object: a specific instance created from the blueprint
  - Example: `water` and `caffeine` objects created from `Molecule` class
  - Each holds its own unique data

Why use classes?

Scripts with loose variables are error-prone. Classes provide:

- Single, coherent object (e.g., `water`) that knows its own data
- Call methods on the object (e.g., `water.calculate_center_of_mass()`)
- Bundle data and behavior together
- Safer, more organized, easier to debug and share

The Basic Structure: `class` and `__init__`

Basic components:

- `class` keyword followed by class name
- Methods: functions that belong to a class
- `__init__`: the constructor
  - Called automatically when creating new instance
  - Sets up initial state by creating attributes

```
class Molecule:
    # __init__ is the constructor method.
    # It runs when you create a new Molecule object.
    def __init__(self, name, charge, coordinates):
        print(f"Creating a new Molecule object for {name}...")

        # --- Attributes ---
        # These are variables that belong to the object.
        # They are created by assigning to `self`.
        self.name = name
        self.charge = charge
        self.coordinates = coordinates
```

The `self` Keyword: The Most Important Concept

`self` is the first argument of any instance method.

Key points:

- Refers to the specific object (instance) the method is called on
- When you write `my_molecule.some_method()`, Python automatically passes `my_molecule` as `self`

Why is `self` needed?

- Methods are defined once on the class (blueprint)
- Multiple objects can exist (`water`, `caffeine`)
- When calling `water.calculate_center_of_mass()`, Python needs to know which object's data to use
- `self` makes that connection

Usage:

- Access object's own attributes and methods inside the method
- Example: `self.name = name` stores input `name` in this specific object's attribute

Building a Class: A Step-by-Step Example

To understand the syntax, we will build up a single, coherent `Molecule` class from scratch.

1. The `__init__` Method and Instance Attributes

Goal: Create `Molecule` objects with unique `name`, `formula`, and `coordinates`.

- Instance attributes: unique to each object
- Class attribute: `_known_elements` set, shared across all `Molecule` objects

```python
import numpy as np

class Molecule:
    # This is a CLASS ATTRIBUTE, shared by all instances.
    _known_elements = set()

    def __init__(self, name, formula, coordinates):
        # --- INSTANCE ATTRIBUTES ---
        # These belong to the specific object being created (`self`).
        self.name = name
        self.formula = formula
        self.coordinates = np.array(coordinates)

        # Update the shared class data based on this instance's data
        elements_in_formula = ''.join(filter(str.isalpha, formula))
        for element in elements_in_formula:
            self.__class__._known_elements.add(element)
```

2. Instance Methods

Instance methods work with unique data of each object. All take `self` as first argument for access to object attributes.

```python
# Continuing the Molecule class...
    def calculate_center_of_mass(self):
        """Calculates the center of mass for this specific molecule."""
        return np.mean(self.coordinates, axis=0)

    def move(self, vector):
        """Moves this specific molecule by a given vector."""
        self.coordinates += vector
```

3. Class Methods

Class methods work with shared data across all instances. Take `cls` as first argument for access to class attributes. Can also serve as alternative constructors.

```
# Continuing the Molecule class...
@classmethod
def get_all_known_elements(cls):
    """Returns the set of elements seen across all molecules."""
    return sorted(list(cls._known_elements))

@classmethod
def from_pdb_file(cls, filename):
    """Creates a Molecule instance by reading a PDB file."""
    # In a real function, you would parse the file.
    # Here, we'll just use placeholder data.
    print(f"Parsing {filename}...")
    parsed_name = "Caffeine"
    parsed_formula = "C8H10N4O2"
    parsed_coords = np.random.rand(24, 3) # Placeholder

    # The method calls the standard constructor `cls(...)` to create the object.
    return cls(parsed_name, parsed_formula, parsed_coords)
```

4. Static Methods

Static methods provide utility functions related to the class but don't need access to instance or class data.

```
# Continuing the Molecule class...
@staticmethod
def get_atomic_mass(element_symbol):
    """A utility function to get atomic mass."""
    masses = {'H': 1.008, 'C': 12.011, 'O': 15.999, 'N': 14.007}
    return masses.get(element_symbol, 0.0)
```

Usage:

```
# Call static method on the class itself
mass = Molecule.get_atomic_mass('C')  # Returns 12.011

# Can also call on an instance (but not recommended)
water = Molecule('Water', 'H2O', [[0,0,0]])
mass = water.get_atomic_mass('H')  # Returns 1.008
```

The Complete Class

Here is the full, coherent `Molecule` class we have built. All the pieces work together.

```python
import numpy as np

class Molecule:
    # Class Attribute (shared)
    _known_elements = set()

    # Constructor and Instance Attributes
    def __init__(self, name, formula, coordinates):
        self.name = name
        self.formula = formula
        self.coordinates = np.array(coordinates)
        elements_in_formula = ''.join(filter(str.isalpha, formula))
        for element in elements_in_formula:
            self.__class__._known_elements.add(element)

    # Instance Methods
    def calculate_center_of_mass(self):
        return np.mean(self.coordinates, axis=0)

    def move(self, vector):
        self.coordinates += vector

    # Class Methods
    @classmethod
    def get_all_known_elements(cls):
        return sorted(list(cls._known_elements))

    @classmethod
    def from_pdb_file(cls, filename):
        print(f"Parsing {filename}...")
        parsed_name, parsed_formula, parsed_coords = "Caffeine", "C8H10N4O2", np.random.rand(2
        return cls(parsed_name, parsed_formula, parsed_coords)

    # Static Method
    @staticmethod
    def get_atomic_mass(element_symbol):
        masses = {'H': 1.008, 'C': 12.011, 'O': 15.999, 'N': 14.007}
        return masses.get(element_symbol, 0.0)
```

Summary: Instance vs. Class vs. Static

The key is to distinguish between data that is unique to an instance versus data that is shared across the entire class.

Method Type	Decorator	First Argument	Operates On…
Instance Method	(None)	`self`	Unique instance data (e.g., `self.coordinates`)
Class Method	`@classmethod`	`cls`	Shared class data (e.g., `cls._known_elements`)
Static Method	`@staticmethod`	(None)	Only its own inputs

Naming Conventions

Python doesn't strictly enforce naming rules, but strong community conventions improve readability.

- Class Names: `CamelCase` (first letter of each word capitalized)
  - Examples: `Molecule`, `SimulationAnalysis`
- Function and Method Names: `snake_case` (lowercase with underscores)
  - Examples: `calculate_center_of_mass`, `load_data`
- Variable and Attribute Names: `snake_case`
  - Examples: `self.coordinates`, `carbon_mass`

These conventions make it easy to distinguish classes, functions, and variables at a glance.

Simulating Multiple Constructors (vs. C++)

In C++, you can define multiple constructors with different arguments (function overloading). Python doesn't allow this—a second `__init__` would overwrite the first.

Python offers two patterns to achieve the same goal.

Pattern 1: Default Arguments in `__init__`

Single `__init__` with optional arguments (default to `None`). Use `if/elif` logic to handle different cases.

```
class Molecule:
    def __init__(self, name, coordinates=None, filename=None):
        self.name = name
        if coordinates is not None:
            self.coordinates = np.array(coordinates)
        elif filename is not None:
            # In reality, you would parse the file here
            self.coordinates = np.random.rand(10, 3)
        else:
            raise ValueError("Must provide coordinates or a filename.")

# --- Usage ---
water = Molecule('Water', coordinates=[[0,0,0]])
caffeine = Molecule('Caffeine', filename='caffeine.pdb')
```

Pros: Simple for few variations
Cons: `__init__` can become complex and messy

Pattern 2: `@classmethod` Factories (Recommended)

Most powerful and Pythonic. Keep `__init__` simple and direct. Create separate, well-named class methods for each alternative constructor. Same "alternative constructor" pattern discussed earlier.

```
class Molecule:
    # The main constructor is simple and direct
    def __init__(self, name, coordinates):
        self.name = name
        self.coordinates = np.array(coordinates)

    # This is an ALTERNATIVE constructor
    @classmethod
    def from_file(cls, name, filename):
        # 1. Perform special logic (e.g., read the file)
        coordinates_from_file = np.random.rand(10, 3) # Placeholder
        # 2. Call the main constructor `cls(...)` to create the final object
        return cls(name, coordinates_from_file)

# --- Usage ---
water = Molecule('Water', coordinates=[[0,0,0]])
caffeine = Molecule.from_file('Caffeine', filename='caffeine.pdb')
```

Pros: Clean, readable, scalable. `Molecule.from_file()` is self-documenting
Cons: Requires slightly more lines of code

Advanced Topic: Simulating Multiple Dispatch (vs. Julia)

While C++ uses function overloading, other scientific languages like Julia use a more powerful concept called **multiple dispatch**. This allows the system to choose which function to run based on the runtime types of *all* of its arguments. This is extremely useful for implementing different physical models for different types of interactions.

Python does not have this feature built-in, but you can achieve the same effect with an external library called `multipledispatch`.

The Problem: Different Interactions

Imagine you have different particle types, and the `interact` function should change based on which types it receives.

```
class Atom: pass
class Ion: pass

# interact(Atom(), Atom())   -> should be Lennard-Jones
# interact(Ion(), Ion())     -> should be Coulomb
# interact(Atom(), Ion())    -> should be Polarization
```

The Solution: `multipledispatch`

First, install the library: `pip install multipledispatch`. Then, you can define your functions in a clean, declarative way.

```
from multipledispatch import dispatch

class Atom: pass
class Ion: pass

@dispatch(Atom, Atom)
def interact(p1, p2):
    print("Calculating Lennard-Jones potential...")

@dispatch(Ion, Ion)
def interact(p1, p2):
    print("Calculating Coulomb potential...")

@dispatch(Atom, Ion)
def interact(p1, p2):
    print("Calculating polarization model...")

@dispatch(Ion, Atom)
def interact(p1, p2):
    # Handle commutative cases by calling the other version
    return interact(p2, p1)

# --- Usage ---
# The library dispatches to the correct function based on the types
interact(Atom(), Atom())
interact(Ion(), Ion())
interact(Atom(), Ion())
```

This pattern is far superior to a messy `if/isinstance` chain. It keeps your scientific logic clean, readable, and easily extensible.

Key Takeaways

- A **Class** is a blueprint for creating objects. An **Object** is a specific instance created from that blueprint, bundling its own unique data (attributes) and behaviors (methods).
- The `__init__` method is the **constructor**, responsible for setting up an object's initial state by assigning values to `self`.
- **Instance Methods** use `self` to operate on the unique data of a specific object.
- **Class Methods** use `@classmethod` and `cls` to operate on shared data that belongs to the class blueprint itself.
- **Static Methods** use `@staticmethod` for utility functions that are related to the class but don't depend on any instance or class state.
- Python does not have C++-style multiple constructors, but the same effect is achieved more cleanly using **`@classmethod` factories**.

This syntax provides the complete toolkit for defining the behavior of your objects, from instance-specific actions to class-level operations and related utilities.

Quick Quiz

Test your understanding of the key concepts.

1. What is the primary difference between a class and an object (or instance)?

Answer

A **class** is the blueprint or template. It defines the structure and behavior. An **object** is a specific, concrete instance created from that blueprint, holding its own unique data.

2. You have a `Simulation` class. You want to add a method that calculates the kinetic energy, which depends on the unique velocities of the particles in that specific simulation. What kind of method should you write?

Answer

You should write an **instance method**. Its first argument will be `self`, allowing it to access `self.velocities` to perform the calculation for that specific simulation instance.

3. You want to add a method to your `Simulation` class that keeps a running count of how many total simulations have been created. What kind of method should you use to retrieve this count?

Answer

You should use a **class method**. You would store the counter in a *class attribute* (e.g., `_simulation_count = 0`). The `__init__` method of each instance would increment this counter (`self.__class__._simulation_count += 1`). The class method (e.g., `get_total_count()`) would then access this shared counter via its `cls` argument (`return cls._simulation_count`).

4. You want to add a function to your `Simulation` class that converts a temperature from Kelvin to Celsius. This calculation is logically related to simulations, but does not depend on any specific simulation's data or on the class itself. What is the most appropriate method type?

Answer

A **static method** (using `@staticmethod`). It doesn't need `self` or `cls`, making it the perfect choice for a self-contained utility function that is namespaced within the class.

Python Classes: Inheritance

Building a Layered System

Research projects grow complex. Climate models have layered, interacting components: atmospheric, ocean, sea ice models. Bioinformatics pipelines chain distinct tools: alignment, variant calling, annotation.

Flat collections of functions and scripts don't represent these layered systems well. Hard to see relationships. Hard to reuse or replace parts without breaking everything.

Inheritance is the primary tool for managing this complexity. Build a hierarchy of classes that mirrors your scientific problem's logical structure. Define general, abstract concepts at the top (e.g., `ForceField`). Create specific, concrete implementations below (e.g., `LennardJonesForceField`, `CoulombForceField`).

Two main goals:

1. Build a logical hierarchy: code structure reflects problem structure

2. Maximize code reuse: write common logic once in parent class, reuse in all child classes

The Problem: Code Duplication

Imagine you are writing a simulation. You start by creating a class for an `Atom`.

```python
class Atom:
    def __init__(self, mass, position, velocity):
        self.mass = mass
        self.position = np.array(position)
        self.velocity = np.array(velocity)

    def advance_position(self, dt):
        self.position += self.velocity * dt
```

Now, you need to add `Ions` to your simulation. An ion is just like an atom, but it also has a charge. Without inheritance, your first instinct might be to copy-paste:

```python
# The copy-paste approach (BAD PRACTICE)
class Ion:
    def __init__(self, mass, position, velocity, charge):
        self.mass = mass
        self.position = np.array(position)
        self.velocity = np.array(velocity)
        self.charge = charge # The only new line!

    def advance_position(self, dt):
        # This method is identical to the one in Atom!
        self.position += self.velocity * dt
```

This is a recipe for disaster. You now have duplicated code. If you find a bug in `advance_position`, you have to remember to fix it in two places. This makes your code harder to maintain and less reproducible. Inheritance solves this problem by allowing you to define the common logic once and reuse it.

Parent and Child Classes

Inheritance formalizes the relationship between your concepts.

- Parent class (base class, superclass): contains all common data and methods
  - Example: `Particle` class
- Child classes (derived classes, subclasses): inherit all attributes and methods from parent
  - Can add unique attributes and methods
  - Can modify inherited ones

Key concept: the “is-a” relationship. An `Ion` is a `Particle`. An `Atom` is a `Particle`. Anything you can do to a `Particle`, you can do to an `Ion`.

Syntax and Examples

Let’s rebuild our example the right way.

1. Creating the Parent Class

First, we define the general `Particle` class with all the shared logic.

```
import numpy as np

class Particle:
    """The parent class for all simulation particles."""
    def __init__(self, mass, position, velocity):
        self.mass = mass
        self.position = np.array(position, dtype=float)
        self.velocity = np.array(velocity, dtype=float)

    def advance_position(self, dt):
        """Advances the particle's position based on its velocity."""
        self.position += self.velocity * dt

    def calculate_kinetic_energy(self):
        """Calculates the kinetic energy."""
        return 0.5 * self.mass * np.dot(self.velocity, self.velocity)
```

2. Creating Child Classes

Now, we can create specialized child classes. The syntax is `class ChildClassName(ParentClassName):`.

```
class Atom(Particle):
    """An Atom is a type of Particle. It inherits everything."""
    # We don't need to write anything here yet!
    # It automatically gets __init__, advance_position, etc. from Particle.
    pass

class Ion(Particle):
    """An Ion is a Particle that also has a charge."""
    def __init__(self, mass, position, velocity, charge):
        # --- Calling the Parent's Constructor ---
        # It's crucial to initialize the parent class first.
        # super() refers to the parent class (Particle).
        super().__init__(mass, position, velocity)

        # Now, add the new attribute that is unique to the Ion.
        self.charge = charge
```

The `super().__init__(...)` line is critical. It says, “run the `__init__` method of my parent class,” which handles setting up the `mass`, `position`, and `velocity`. We then only need to add the one new line to handle the `charge`.

3. Using the Inherited Classes

```
# Create an instance of the Atom child class
argon_atom = Atom(mass=39.9, position=[0,0,0], velocity=[1,0,0])

# Create an instance of the Ion child class
```

```
sodium_ion = Ion(mass=22.9, position=[5,5,5], velocity=[0,1,0], charge=1)

# We can call the method defined in the PARENT class on both objects
argon_atom.advance_position(0.1)
sodium_ion.advance_position(0.1)

print(f"Argon atom new position: {argon_atom.position}")
print(f"Sodium ion new position: {sodium_ion.position}")

# The sodium_ion object has the extra attribute
print(f"Sodium ion charge: {sodium_ion.charge}")
```

4. Overriding and Extending Methods

What if we want a child class to behave differently? We can **override** a parent's method by simply defining a method with the same name in the child class.

Let's add a `describe` method.

```
class Particle:
    # ... (previous methods) ...
    def describe(self):
        return f"A particle with mass {self.mass:.2f}."

class Ion(Particle):
    # ... (__init__ from above) ...

    # This OVERRIDES the parent's describe method
    def describe(self):
        # But we can still call the parent's version first using super()
        # This is called EXTENDING the method.
        parent_description = super().describe()
        return f"{parent_description} And it has a charge of {self.charge}."

# --- Comparing the behavior ---
p = Particle(1.0, [0,0,0], [0,0,0])
ion = Ion(22.9, [5,5,5], [0,1,0], 1)

print(p.describe())
print(ion.describe())
```

Output:

```
A particle with mass 1.00.
A particle with mass 22.90. And it has a charge of 1.
```

The `Ion` class first calls the original `describe` method from the `Particle` class using `super().describe()` and then *extends* it by adding its own specific information.

By using inheritance, you create a logical and intuitive structure for your code that eliminates duplication and makes your scientific models easier to build, maintain, and share.

See the complete code: `src/atom.py`.

Abstract Classes: Defining a Contract (The Python Equivalent of C++ Virtual Classes)

Core idea: create a "contract" or "template" for other classes to follow. Define a general parent class that should not be used on its own, but serves as a blueprint guaranteeing all child classes have consistent structure and specific methods. Cornerstone of building large, reliable software systems.

C++ does this with "abstract base classes" and "pure virtual functions." Python provides the same functionality through the built-in `abc` (Abstract Base Class) module.

Defining a "Contract"

Modeling different force fields. Every valid force field must calculate the energy of a system. We want to enforce this rule in code.

Create an abstract `ForceField` class with an "abstract method" called `calculate_energy`. This class acts as a contract. Cannot create an instance of `ForceField` itself. Can only create instances of child classes that fulfill the contract by providing their own concrete implementation of `calculate_energy`.

The Syntax: `ABC` and `@abstractmethod`

```python
from abc import ABC, abstractmethod

# 1. Inherit from ABC to mark this as an abstract base class.
class ForceField(ABC):
    """
    An abstract base class (a contract) for all force fields.
    It cannot be instantiated directly.
    """
    def __init__(self, parameters):
        self.parameters = parameters

    # 2. Use the @abstractmethod decorator.
    # This declares that any concrete child class MUST implement this method.
    @abstractmethod
    def calculate_energy(self, positions):
        """Calculates the potential energy of the system."""
        pass

# --- Now, we create concrete child classes that fulfill the contract ---

class LennardJones(ForceField):
    """A concrete implementation for the Lennard-Jones potential."""

    # We MUST implement this method, or Python will raise an error.
    def calculate_energy(self, positions):
        # In a real scenario, this would be a complex calculation.
        print("Calculating energy using the Lennard-Jones potential...")
        return np.sum(positions**2) # Placeholder

class Coulomb(ForceField):
    """A concrete implementation for the Coulomb potential."""

    # We also MUST implement this method.
    def calculate_energy(self, positions):
        print("Calculating energy using the Coulomb potential...")
        return np.sum(np.abs(positions)) # Placeholder
```

Enforcing the Contract

The key feature is that Python will prevent you from creating an object that doesn't follow the rules.

```
# This works perfectly. LennardJones fulfills the contract.
lj_potential = LennardJones(parameters={'sigma': 1.0, 'epsilon': 0.1})
lj_potential.calculate_energy(np.random.rand(10, 3))

# This will FAIL with a TypeError!
# You cannot create an instance of an abstract class.
try:
    invalid_ff = ForceField(parameters={})
except TypeError as e:
    print(f"\nError: {e}")
```

Output:

```
Calculating energy using the Lennard-Jones potential...

Error: Can't instantiate abstract class ForceField with abstract method calculate_energy
```

This is the same behavior you would get from a C++ class with a pure virtual function. The `abc` module is the "Pythonic" way to define a mandatory interface for a family of related classes, which is a cornerstone of robust, large-scale software design.

Multi-Level Inheritance

Inheritance is not limited to a single parent-child level. You can build entire family trees of classes, where a child class becomes a parent to a grandchild class. This allows you to create highly specialized classes that inherit and combine features from their entire ancestry.

A Simple Example: The Animal Kingdom

Let's model a simple hierarchy: `Animal` -> `Mammal` -> `Dog`.

1. **The Base Class**

   This class is very general. It has a feature common to all animals: they have an age.

   ```
   class Animal:
       def __init__(self, age):
           self.age = age

       def report_age(self):
           return f"I am {self.age} years old."
   ```

2. **The Parent Class (Intermediate Class)**

This class inherits from Animal. It adds a feature common to all mammals: they have fur.

```python
class Mammal(Animal):
    def __init__(self, age, fur_color):
        # Initialize the parent class to set the age
        super().__init__(age)
        self.fur_color = fur_color

    def describe_fur(self):
        return f"I have {self.fur_color} fur."
```

3. **The Child Class (Concrete Class)** This class inherits from Mammal. It adds a feature unique to dogs: they can bark.

```python
class Dog(Mammal):
    def __init__(self, age, fur_color, breed):
        # Initialize the parent class (Mammal)
        # The Mammal's __init__ will in turn initialize the Animal class.
        super().__init__(age, fur_color)
        self.breed = breed

    def bark(self):
        return "Woof!"
```

Using the Grandchild Class

An instance of the Dog class now has access to the methods and attributes from its entire inheritance chain: Dog, Mammal, and Animal.

```python
my_dog = Dog(age=5, fur_color='brown', breed='Golden Retriever')

# Method from the Dog class
print(my_dog.bark())

# Method from the Mammal class
print(my_dog.describe_fur())

# Method from the Animal class
print(my_dog.report_age())

# Attributes from all levels
print(f"Breed: {my_dog.breed}, Fur: {my_dog.fur_color}, Age: {my_dog.age}")
```

Output:

```
Woof!
I have brown fur.
I am 5 years old.
Breed: Golden Retriever, Fur: brown, Age: 5
```

This layered approach is extremely powerful for organizing complex scientific models. You can define general physical laws in a base class, add more specific behaviors in intermediate classes, and create highly specialized, concrete classes for your final simulations, all while maximizing code reuse and maintaining a clear, logical structure.

Python's Inheritance vs. C++ and Julia

Coming from other scientific computing languages? Python's inheritance is simpler and more flexible—significant advantage for rapid research and development.

- Compared to C++: Python's inheritance is much less verbose
  - No Header Files: define class and methods in single `.py` file
  - No `virtual` Keyword: all methods are "virtual" by default. When you call `my_object.my_method()`, Python runs the version from the most specific child class. No explicit declaration needed
  - Multiple Inheritance: Python supports inheriting from multiple parents (`class C(A, B):`). Powerful but complex. C++ supports this too, but Python's dynamic nature makes it easier to manage
- Compared to Julia: Julia's design is fundamentally different
  - Composition over Inheritance: Julia strongly favors "composition" over inheritance. Instead of `Dog` being an `Animal`, create a `Dog` struct that has an `Animal` struct inside it
  - No Shared Behavior: cannot inherit methods in Julia (can inherit types). Primary mechanism for sharing behavior is multiple dispatch—define functions that operate on different data types

In Julia, this would look like:

```julia
# You can define a type hierarchy
abstract type Animal end

struct Dog <: Animal  # Dog is a subtype of Animal
    name::String
end

struct Cat <: Animal
    name::String
end

# methods are defined OUTSIDE the types
function speak(a::Animal)
    println("Some generic animal sound")
end

function speak(d::Dog)
    println("$(d.name) says Woof!")
end

function speak(c::Cat)
    println("$(c.name) says Meow!")
end
```

The Advantage for Scientific Prototyping:

- Python hits a sweet spot for scientific computing
- Inheritance model is powerful enough to build logical hierarchies needed for complex models
- Avoids the rigid, boilerplate-heavy syntax of C++
- Faster to prototype, easier to read, and more straightforward to refactor

- Flexibility is crucial as your scientific understanding of the problem evolves

Polymorphism in Scientific Computing

All the concepts about inheritance, method overriding, and abstract classes build towards a powerful design principle: polymorphism. Write flexible, high-level code that operates on a wide variety of different objects in a uniform way.

Core idea: single function accepts objects of different classes, as long as they all "look" the same by adhering to a common interface.

The Problem

Run the same simulation pipeline, but test three different force field models: `LennardJones`, `Coulomb`, and a new `AI_Potential`. Without polymorphism, you might write an `if/elif/else` block:

```python
# The NON-polymorphic way (BAD PRACTICE)
def run_simulation(force_field, positions):
    if isinstance(force_field, LennardJones):
        energy = force_field.calculate_energy(positions)
    elif isinstance(force_field, Coulomb):
        energy = force_field.calculate_energy(positions)
    # ... you have to modify this function every time you add a new model!
```

This is not scalable and is hard to maintain because:

- You must explicitly check the type with `isinstance()`
- You must modify this function every time you add a new force field class!
- The function is tightly coupled to specific class names

The Polymorphic Solution

By using the `ForceField` abstract base class we defined earlier, we create a contract that all force fields must follow. This allows us to write a single, clean, high-level function that doesn't need to know the specific details of the model it's working with.

```python
# This is a POLYMORPHIC function.
# It can accept any object that fulfills the ForceField contract.
def run_simulation(force_field: ForceField, positions):
    """Runs a simulation step using a given force field."""
    print(f"\n--- Running simulation with {force_field.__class__.__name__} ---")
    # It doesn't know the specific type, it just trusts the contract.
    energy = force_field.calculate_energy(positions)
    print(f"Calculated Energy: {energy:.2f}")

# --- Now we can pass in different types of objects ---
lj = LennardJones(parameters={})
coulomb = Coulomb(parameters={})
# Imagine we wrote a third class, AI_Potential, that also inherits from ForceField

run_simulation(lj, np.random.rand(10,3))
run_simulation(coulomb, np.random.rand(10,3))
# run_simulation(AI_Potential(model_file='...'), positions)
```

So polymorphism in Python is more about design philosophy (trusting interfaces) than syntax. The syntax difference is simply: with or without `isinstance()` checks!

This is the ultimate benefit of object-oriented design. Your high-level scientific workflow is now completely **decoupled** from the specific models you are using. You can develop and test new `ForceField` models without ever having to change a single line of your main `run_simulation` function. This makes your research more flexible, your code more robust, and your science more reproducible.

Key Takeaways

- Inheritance: primary tool for building logical hierarchy of classes and maximizing code reuse
- Child class inherits all methods and attributes from parent class. Use `class Child(Parent):` to define this relationship
- Use `super().__init__(...)` in child's constructor to ensure parent class is properly initialized
- Child class can override parent's method by defining method with same name. Can extend parent's method by calling `super().method_name()` inside new implementation
- Abstract Base Classes (using `ABC` and `@abstractmethod`) define a "contract" or interface, forcing child classes to implement specific methods. Python equivalent of C++ pure virtual functions
- Polymorphism: the payoff for using inheritance correctly. Write clean, high-level functions that operate on any object adhering to a common interface, making code flexible and extensible

---

Quick Quiz

1. What are the two primary benefits of using inheritance?

Answer

1. **Code Reuse:** It allows you to write common logic once in a parent class and reuse it in multiple child classes, avoiding code duplication.
2. **Logical Hierarchy:** It allows you to structure your code in a way that reflects the natural, hierarchical relationships of the concepts in your scientific model.

2. You have a parent class `Experiment` and a child class `CalorimetryExperiment`. Both have an `__init__` method. What is the command you must use in the child's `__init__` method to ensure the parent's `__init__` method is also run?

Answer

You must call `super().__init__(...)`, passing along any arguments the parent's constructor needs. This ensures that the setup logic in the parent class is executed before the child class adds its own specific attributes.

3. You are writing a high-level function that needs to work with different types of scientific data readers (e.g., `PDBReader`, `CSVReader`, `HDF5Reader`). You want to guarantee that every reader object has a `.read_data()` method. What is the best way to enforce this contract?

Answer

The best way is to create an **abstract base class** called `DataReader` that inherits from `ABC`. Inside this class, you would define an abstract method: “‘python from abc import ABC, abstractmethod

class DataReader(ABC): @abstractmethod def read_data(self): pass “You would then make sure thatPDBReader,CSVReader, andHDF5Readerall inherit fromDataReaderand provide their own concrete implementations of theread_data‘ method.

4. What is polymorphism?

Answer

Polymorphism is the ability to write a single, high-level function that can operate on objects of different classes. This is possible as long as those different objects all share a common interface (e.g., they all inherit from the same base class or have methods with the same names). It allows you to decouple your main workflow from the specific implementations of your models, making your code more flexible and extensible.

Modern Python Classes: Dataclasses

This section covers `@dataclass`: a decorator that can simplify your class definitions.

The Power of `@dataclass`

Now that you understand decorators, we can introduce one of the most useful decorators for scientific programming: `@dataclass`.

We have learned how to write a class manually, including the `__init__` method to store attributes. This involves a lot of boilerplate code.

The “Before” Picture: A Manual Class

```python
class ManualMolecule:
    def __init__(self, name, charge, num_atoms):
        self.name = name
        self.charge = charge
        self.num_atoms = num_atoms

    def __repr__(self):
        # We have to write our own representation method for printing.
        return f"ManualMolecule(name='{self.name}', charge={self.charge}, num_atoms={self.num_

    def __eq__(self, other):
        # We have to write our own equality method for comparing.
        if not isinstance(other, ManualMolecule):
            return False
        return (self.name == other.name and
                self.charge == other.charge and
                self.num_atoms == other.num_atoms)
```

This is a lot of code just to create a simple data container.

The “After” Picture: Using `@dataclass`

The @dataclass decorator (from the built-in dataclasses module) can write all of that boilerplate for you automatically. All you have to do is declare the attributes using type hints.

```
from dataclasses import dataclass

@dataclass
class Molecule:
    # Just declare the attributes and their types.
    # That's it!
    name: str
    charge: int
    num_atoms: int

# --- Let's see what we get for free ---

# 1. A proper __init__ method is automatically generated.
water = Molecule(name='Water', charge=0, num_atoms=3)

# 2. A beautiful __repr__ method is generated for easy debugging.
print(water)
# Output: Molecule(name='Water', charge=0, num_atoms=3)

# 3. A proper __eq__ method is generated for value-based comparison.
water2 = Molecule(name='Water', charge=0, num_atoms=3)
print(f"Are the two molecules equal? {water == water2}")
# Output: Are the two molecules equal? True
```

Why @dataclass is Perfect for Scientific Computing

In research, we often work with simple objects whose primary purpose is to hold data (e.g., simulation result, experimental parameters, record from a file). @dataclass is the perfect tool for this:

1. Reduces boilerplate: define clean, readable data object in just a few lines
2. Provides useful __repr__: automatic representation makes data objects easy to inspect and debug
3. Makes testing easy: automatic __eq__ method allows easy comparison of two data objects to see if they hold same values—invaluable when writing tests

Using @dataclass lets you write clearer, more concise, and more robust code. Focus on the science instead of writing boilerplate methods.

Python Packaging: Turning Your Code into an Installable Tool

So far, we've focused on structuring code using classes to make it robust and reproducible. But how do you share that code with a colleague or the wider scientific community? Emailing a .py file is a start, but not a professional or reliable solution. Recipient won't know what dependencies are needed, what versions are compatible, and they'll have to manage the file manually.

This is where packaging comes in. Goal: bundle your Python code into a standardized format that can be easily installed, managed, and distributed. Gold standard: make your tool installable with a simple command: pip install your-tool-name.

This section walks you through this process in three parts:

1. Part 1: Creating Your First Package. Turn a local script into a properly structured project you can install and test on your own machine
2. Part 2: Sharing Your Package. Two common ways to distribute your package: via private GitHub repository or public Python Package Index (PyPI)
3. Part 3: The Development Cycle. How to release new versions of your package as you make improvements

Recall: Classes vs. Modules in Python

Classes:

- A class defines a blueprint for creating objects that bundle data (attributes) and behavior (methods).
- Ideal for representing complex entities or workflows where state and operations are tightly coupled (e.g., a simulation, a data processor).
- Supports encapsulation, inheritance, and polymorphism, enabling more advanced software design.
- Classes are defined within modules and can be reused across projects.
- Classes bundle related data and methods for more complex tasks.

Modules:

- A module is a single Python file, or a collection of files, that groups related functions, variables, and classes.
- Useful for organizing code by topic, workflow, or functionality (e.g., `math.py`, `io_utils.py`).
- Promotes code reuse and separation of concerns.
- Modules are imported using the `import` statement, making code easy to share and maintain.

Part 1: Creating Your First Package (v0.1.0)

Our first goal is to take a standalone Python file and turn it into a real, installable package. We will start by creating version `0.1.0`.

Step 1.1: A Standard Project Structure

The Python community has adopted a standard layout for projects that is recognized by all modern packaging tools. The most important convention is to place your actual source code inside a dedicated directory.

Here is the recommended structure:

```
heat1d-project/
   src/
       heat1d/
           __init__.py
           heat1d_class.py
   tests/
       test_heat1d.py
   pyproject.toml
   README.md
```

Let's break this down:

- `heat1d-project/`: The root directory of your project.
- `src/`: A directory that contains your Python source code. This separation is a modern best practice that prevents many common packaging problems.
- `src/heat1d/`: This is your actual **Python package**. The name of this directory is what users will use when they `import` your code.
- `src/heat1d/__init__.py`: This file is often empty. Its presence tells Python that the `heat1d` directory is a package.
- `src/heat1d/heat1d_class.py`: Your actual Python code file.
- `pyproject.toml`: This is the most important file. It's the modern, standardized file for configuring your project, telling Python's build tools everything they need to know to package your code.

Step 1.2: The `pyproject.toml` Configuration File

This file is the heart of your package. It contains all the metadata about your project. Using the TOML (Tom's Obvious, Minimal Language) format, it is designed to be easy for humans to read and write.

Here is a simple but complete `pyproject.toml` for our project:

```
# pyproject.toml

[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "heat1d-simulation"
version = "0.1.0"
authors = [
  { name="Your Name", email="your.email@example.com" },
]
description = "A simple 1D heat equation solver."
readme = "README.md"
requires-python = ">=3.8"
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: MIT License",
    "Operating System :: OS Independent",
]
dependencies = [
    "numpy",
    "matplotlib",
]

[project.urls]
"Homepage" = "https://github.com/your-username/heat1d-project"
"Bug Tracker" = "https://github.com/your-username/heat1d-project/issues"
```

Let's break down the key sections:

- `[build-system]`: This section is mostly boilerplate. It tells `pip` what tools are needed to build your package (in this case, `setuptools`). You can usually just copy-paste this.
- `[project]`: This is where you define your project's metadata.

  - `name`: The name that will be used on PyPI (the Python Package Index). This is what people will type when they run `pip install ...`.
  - `version`: The current version of your package. You will update this number whenever you release a new version.
  - `dependencies`: This is a critical field. It lists all the other packages that your code needs to run (like `numpy` and `matplotlib`). When a user installs your package, `pip` will automatically find and install these dependencies for them.

Step 1.3: Building Your Package

Once you have your project structure and your `pyproject.toml` file, you are ready to build the package. This process creates the actual files that will be distributed.

First, you need to install the standard Python build tool:

```
pip install build
```

Now, from the root of your project directory (`heat1d-project/`), run the build command:

```
python -m build
```

This command creates a new directory called `dist/`. Inside, you'll find two important files:

- `heat1d_simulation-0.1.0-py3-none-any.whl`: wheel file. Pre-built, binary distribution format that makes installation very fast for end-user. Preferred format
- `heat1d_simulation-0.1.0.tar.gz`: source distribution (sdist). Compressed archive of your source code. `pip` can use this as fallback if there isn't a compatible wheel file available

Step 1.4: Installing and Testing Your Package Locally

Before you share your package with the world, you should test it on your own machine. You can install the wheel file you just created directly using `pip`.

Navigate out of your project directory and install it:

```
cd ..
pip install heat1d-project/dist/heat1d_simulation-0.1.0-py3-none-any.whl
```

To overwrite an existing installation, use

```
pip install --force-reinstall dist/heat1d_simulation-0.1.0-py3-none-any.whl
```

Your package is now installed in your Python environment, just like any other package. You can now open a Python interpreter or a script from any directory on your system and use it:

```
python run-heat.py
```

At this point, you have a fully working local package. The next step is to share it with others.

Releasing Your Package

Once your package is built and tested, you need to host the distribution files (`.whl` and `.tar.gz`) somewhere others can access them. Two common methods:

Option A: Distributing via GitHub Releases

For many projects, especially internal tools or research code, full release to PyPI is unnecessary. Great alternative: use GitHub Releases. This workflow ties your package versions directly to your source control—excellent for reproducibility.

Process integrates `git` with the build process.

1. Commit Your Changes: make sure all your code changes for the new version are committed to your repository

2. **Tag the Version:** Create a `git` tag that matches the version number in your `pyproject.toml`. This permanently marks that specific commit as that version. It's common to use tags like `v0.1.0`.

   ```
   # Create a tag for version 0.1.0
   git tag v0.1.0

   # Push the tag to GitHub
   git push origin v0.1.0
   ```

3. **Create a GitHub Release:**

   - Go to your repository on GitHub.
   - Click on the "Releases" link on the right-hand side.
   - Click "Draft a new release".
   - In the "Choose a tag" dropdown, select the tag you just pushed (e.g., `v0.1.0`).
   - Give the release a title (e.g., "Version 0.1.0") and write a description of the changes.
   - In the "Attach binaries" section, drag and drop the `.whl` and `.tar.gz` files you created in the `dist/` directory.
   - Click "Publish release".

Your package is now available on GitHub.

How Users Install from GitHub

A user can now install your package directly from your GitHub release using `pip`. They just need the URL to the wheel file.

1. The user goes to your GitHub Releases page.

2. They find the release they want (e.g., `v0.1.0`).

3. They right-click on the `.whl` file and copy its URL.

4. They use that URL with `pip install`:

   ```
   pip install https://github.com/your-username/heat1d-project/releases/download/v0.1.0/heat
   ```

   `pip` will download the file and install it along with any dependencies, just as if it were coming from PyPI. This method is a powerful way to share installable versions of your code without needing a public package index.

Option B: Distributing on the Python Package Index (PyPI)

The final step for sharing your work with the entire community is to upload it to the Python Package Index (PyPI). This is the official central repository where `pip` looks for packages.

1. **Create an Account:** Register for an account on pypi.org.

2. **Install the Upload Tool:**

   ```
   pip install twine
   ```

3. **Upload Your Distribution Files:**

   ```
   twine upload dist/*
   ```

   `twine` will prompt you for your PyPI username and password.

Once the upload is complete, your package is live! Anyone in the world can now install it by simply running:

```
pip install heat1d-simulation
```

Part 3: The Development Cycle: Releasing a New Version

Once your package is released, you will inevitably want to improve it. The process of releasing an update is a cycle of coding, versioning, and re-distributing.

Let's say you want to release version `0.2.0`.

1. **Make Your Code Changes:** Add your new features or fix bugs in the source code.
2. **Update the Version Number:** This is a critical step. Open your `pyproject.toml` file and increment the `version` number.

   ```
   - version = "0.1.0"
   + version = "0.2.0"
   ```

   It's common to follow a system called Semantic Versioning (Major.Minor.Patch).

3. **Re-run the Build:**

   ```
   python -m build
   ```

   This will create new `heat1d_simulation-0.2.0-py3-none-any.whl` and `heat1d_simulation-0.2.0.tar.gz` files in your `dist/` directory.

4. **Distribute the New Files:** Repeat the steps from Part 2 to share your new version.
   - **For GitHub:** Create a new git tag (`v0.2.0`), push it, and create a new GitHub Release, uploading the new distribution files.
   - **For PyPI:** Run `twine upload dist/*` again to upload the new version. PyPI will automatically handle the version update.

This cycle ensures that users can reliably access specific versions of your code and understand what has changed between releases.

Key Takeaways

- **Packaging** is the process of turning your Python code into a standardized, installable format.
- The process can be broken down into three stages: **creating** the package locally, **sharing** it via a platform like GitHub or PyPI, and **updating** it by releasing new versions.
- A modern Python project should use a `src/` layout and be configured with a `pyproject.toml` file, which defines its name, version, and dependencies.
- Use the `build` library (`python -m build`) to create distribution files (wheels and source distributions).
- Always test your package locally with `pip` before distributing it.

- To release a new version, you must update the `version` number in `pyproject.toml`, re-build, and then re-distribute the new files.

By following these steps, you can transform your scientific scripts into professional, reusable, and easily shareable software packages. This is a critical skill for ensuring your computational research is reproducible and accessible to others.

Calling External Code from Python: A Scientist's Guide

As computational scientists, we often stand on the shoulders of giants. Many powerful simulation engines and numerical libraries are written in compiled languages like C, Fortran, or modern high-performance languages like Julia. This tutorial explains the standard Python tools for interfacing with this external code.

Part 1: Calling Compiled C & Fortran with `ctypes`

The most common task is calling functions from existing, compiled shared libraries (`.so`, `.dll`, `.dylib`). Python's built-in **`ctypes`** library is the perfect tool for this. It acts as a direct, low-level bridge to these libraries without requiring any extra installation or compilation steps for your Python code.

The "Why": When to Use `ctypes`

Use `ctypes` when you have a pre-compiled library and you want to call functions from it directly, especially when you want to avoid setting up a complex build system.

Practical Example 1: Calling C

Let's start with a simple C function.

1. The C Code (`libadd.c`)

```c
double add_doubles(double a, double b) {
    return a + b;
}
```

2. Compile to a Shared Library

```bash
gcc -shared -fPIC -o libadd.so libadd.c
```

3. The Python `ctypes` Code The Python script must load the library, define the function's argument and return types, and then call it.

```python
import ctypes
import os

# 1. Load the library
lib_path = os.path.join(os.path.dirname(__file__), 'libadd.so')
my_c_library = ctypes.CDLL(lib_path)

# 2. Define the function signature
add_doubles_func = my_c_library.add_doubles
add_doubles_func.argtypes = [ctypes.c_double, ctypes.c_double]
add_doubles_func.restype = ctypes.c_double

# 3. Call the function
result = add_doubles_func(10.5, 20.2)
print(f"Result from C library: {result}")
```

Practical Example 2: Calling Fortran

Calling Fortran is similar, but with two key differences: **name mangling** (the function name is often changed to lowercase with a trailing underscore) and **pass-by-reference** (all arguments are passed as pointers).

1. The Fortran Code (`libadd_f.f90`)

```
subroutine add_doubles_f(a, b, result)
    real(8), intent(in) :: a, b
    real(8), intent(out) :: result
    result = a + b
end subroutine add_doubles_f
```

2. Compile to a Shared Library

```
gfortran -shared -fPIC -o libadd_f.so libadd_f.f90
```

3. The Python ctypes Code

```
import ctypes
import os

# 1. Load the library
lib_path = os.path.join(os.path.dirname(__file__), 'libadd_f.so')
my_f_library = ctypes.CDLL(lib_path)

# 2. Define the function signature (note the name change and pointer types)
add_doubles_func = my_f_library.add_doubles_f_
add_doubles_func.argtypes = [
    ctypes.POINTER(ctypes.c_double),
    ctypes.POINTER(ctypes.c_double),
    ctypes.POINTER(ctypes.c_double)
]
add_doubles_func.restype = None

# 3. Prepare arguments and call by reference
a = ctypes.c_double(10.5)
b = ctypes.c_double(20.2)
result = ctypes.c_double()
add_doubles_func(ctypes.byref(a), ctypes.byref(b), ctypes.byref(result))

print(f"Result from Fortran library: {result.value}")
```

Part 2: Bridging High-Level Languages

When interfacing with other modern scientific languages like Julia or R, `ctypes` is not the right tool. Instead, we use dedicated high-level libraries that manage the communication between the two language runtimes.

Calling Julia from Python with `PyJulia`

The `julia` library (`pip install julia`) starts a Julia session in the background and provides a seamless bridge.

1. The Julia Code (`my_analysis.jl`)

```julia
function process_data(data, constant)
    return data.^2 .+ constant
end
```

2. The Python Code

```python
import numpy as np
from julia import Main as jl

# This starts the Julia VM (can be slow the first time)
jl.include("my_analysis.jl")

python_array = np.array([1.0, 2.0, 3.0])
# PyJulia handles the conversion of NumPy arrays automatically
result_array = jl.process_data(python_array, 100.0)
print(f"Result from Julia: {result_array}")
```

Calling R from Python with `rpy2`

Similarly, the `rpy2` library (`pip install rpy2`) embeds an R process in Python, with excellent support for converting pandas DataFrames.

```python
import pandas as pd
from rpy2.robjects import pandas2ri
from rpy2.robjects.packages import importr

pandas2ri.activate()
stats = importr('stats')

py_df = pd.DataFrame({'x': [1,2,3,4,5], 'y': [1.1,1.9,3.2,4.3,5.1]})

# rpy2 converts the pandas DataFrame and calls R's lm() function
linear_model = stats.lm("y ~ x", data=py_df)
coefficients = stats.coef(linear_model)
print(f"Slope from R linear model: {coefficients[1]:.4f}")
```

Part 3: Choosing the Right Tool - The Interoperability Ecosystem

`ctypes` is just one tool in a rich ecosystem. Here is a guide to help you choose the right one for your task.

- **To accelerate your Python code:**
  - **Numba:** The best choice for speeding up `for` loops over NumPy arrays using a JIT compiler.
  - **Cython:** A more powerful tool that compiles a Python-like language to C. Use it for complex acceleration tasks or for creating detailed C-level bindings.
- **To call existing C/C++/Fortran libraries:**
  - **`ctypes` (this tutorial):** The best choice when you need a quick, simple way to call a few functions from a pre-compiled library without any extra build steps. It's built-in and easy to use.
  - **`pybind11`:** The modern standard for creating clean, high-quality bindings for C++ libraries. This is the tool of choice if you are a C++ developer looking to expose your code to Python.
  - **Cython:** Can also be used to create very powerful, high-performance bindings, giving you more control than `ctypes`.
- **To interface with other high-level languages:**
  - **`PyJulia`:** The standard for calling Julia.
  - **`rpy2`:** The standard for calling R.

By understanding this landscape, you can choose the most appropriate tool for your specific scientific computing challenge.

Summary and Conclusion

Workshop Overview

Today we covered foundations for sustainable research software:

- Transitioning from linear scripts to well-structured, reusable code
- Using Python functions and classes to organize scientific logic
- Packaging and distributing code for reproducibility
- Building software that scales beyond single-use notebooks

Key Concepts Covered

Functions: Building Blocks of Reusable Code

- Type hints improve readability and enable static analysis
- Function signatures define clear contracts (arguments and return types)
- Exception handling makes code robust and prevents crashes
- Default arguments and keyword-only arguments improve flexibility and clarity
- Proper function design follows single responsibility principle

Classes: Organizing Complex Workflows

- Classes bundle data (attributes) and behavior (methods) into coherent units
- Self keyword connects methods to specific object instances
- Instance methods operate on unique object data
- Class methods work with shared data across all instances
- Static methods provide utility functions without accessing instance or class state
- Inheritance enables code reuse and logical hierarchies
- Abstract base classes define contracts that child classes must fulfill
- Polymorphism allows functions to work with any object adhering to a common interface

Why Classes Matter for Science

- Encapsulation protects data integrity
- Clear workflows make research reproducible
- Safe refactoring enables long-term project evolution
- Multiple independent analyses without variable interference
- Objects act as snapshots of specific analyses

Modern Python Features

- Dataclasses eliminate boilerplate for data-holding classes
- Decorators wrap functions to add behavior (timing, logging, registration)
- Type aliases simplify complex type hints

Packaging: Sharing Your Work

- Standard project structure: `src/` layout with `pyproject.toml`
- Build tool creates wheel and source distributions
- Two distribution methods: GitHub Releases or PyPI
- Version control with git tags ties releases to source control
- Semantic versioning (Major.Minor.Patch) communicates changes clearly

The Development Cycle

Sustainable research software follows a continuous cycle:

1. Write code using functions and classes
2. Organize into modules and packages
3. Test locally before distribution
4. Version and release through GitHub or PyPI
5. Update and iterate as science evolves

Reproducibility: Beyond Code Structure

True reproducibility requires:

- Well-structured code (functions and classes)
- Documented dependencies (requirements.txt, conda environment)
- Version control (git for tracking changes)
- Reproducible environments (conda, Docker, Apptainer)
- Clear installation instructions

From Scripts to Software

The progression we covered:

1. Linear scripts → Functions (single responsibility)
2. Functions → Classes (encapsulation and state)
3. Classes → Modules (organization by topic)
4. Modules → Packages (distribution and reuse)
5. Packages → Released versions (reproducible science)

Why This Matters

- Scripts work once; software works repeatedly
- One-off code cannot be trusted; structured code can be verified
- Floating variables break easily; objects maintain integrity
- Manual installations fail; packages install reliably
- Undocumented dependencies cause "works on my machine" problems

Even in the LLM Era

These skills remain essential:

- You provide the design thinking LLMs need
- You validate and maintain generated code
- You make architectural decisions
- You ensure long-term sustainability

Bottom Line

Sustainable research software requires:

- Functions over linear scripts
- Classes over floating variables
- Packages over emailed files
- Versions over "latest copy"
- Reproducible environments over "works on my machine"

Transform your computational research from fragile scripts into professional, reproducible, shareable software.

Next in This Series

- Workshop 2 (October 16): Scaling your science with parallel computing
- Workshop 3 (October 30): Accelerating your code with GPUs

Thank you for attending today's workshop!