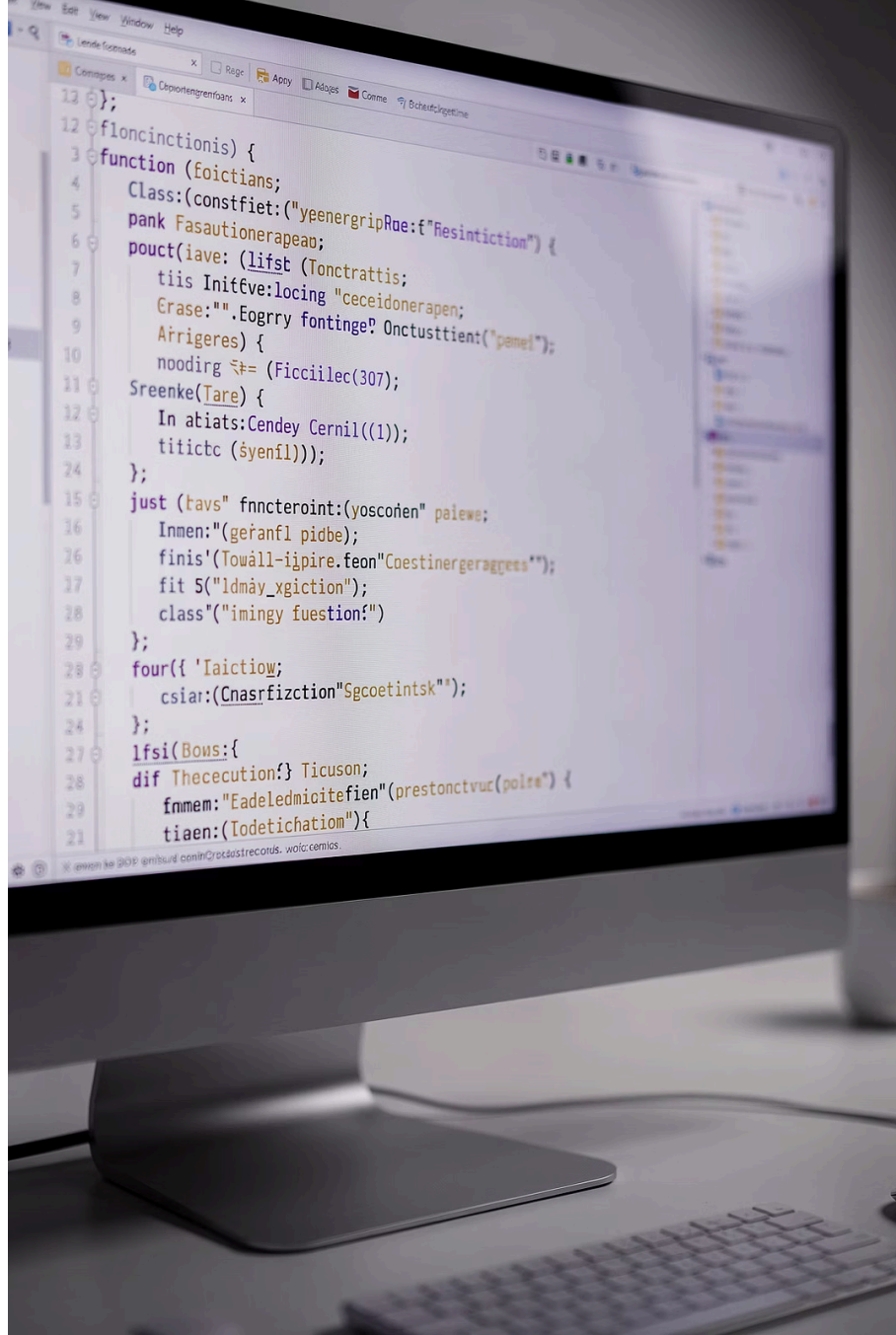
Session 3: OOP Deep Dive

Inheritance, Polymorphism, Interfaces, Properties, Dunder Methods



Learning Objectives

By the end of this session, you will have mastered several advanced object-oriented programming concepts that are essential for building robust, maintainable data applications. These skills will enable you to design flexible systems that can adapt to changing requirements without major rewrites.

01

Inheritance Fundamentals

Explain and use inheritance to create specialised subclasses from base classes, enabling code reuse and establishing clear hierarchies.

02

Polymorphism in Action

Apply polymorphism to write flexible code that works with the same interface but different implementations.

03

Interfaces & Contracts

Define robust interfaces using ABC with @abstractmethod, and understand the role of Protocol for structural typing.

04

Properties & Dunder Methods

Use @property for computed attributes and implement essential dunder methods for Pythonic object behaviour.

05

Design Decisions

Make informed choices between composition and inheritance patterns in real-world data projects.

RECAP

Quick Recap from Session 2

What You Already Know

You've already built a solid foundation in object-oriented programming basics. You understand the distinction between classes (blueprints) and objects (actual instances), and you're comfortable with the fundamental building blocks of Python classes.

- Classes versus objects (instances)
- The __init__ method and self parameter
- Defining attributes and methods
- Building a small, functional object API

Today's Focus

Today we're taking a significant step forward. We're moving beyond working with a single, isolated class into the realm of **reusable, scalable design patterns**. You'll learn how to create systems of interrelated classes that work together elegantly.

Why These OOP Topics Matter

Real-World Data Project Challenges

In professional data projects, you'll rarely work with just one data source or one processing approach. Real-world scenarios demand flexibility and maintainability. Consider the typical challenges data engineers and scientists face daily:

Multiple Data Sources

You need to handle CSV files, JSON documents, API responses, database queries, and more—often within the same project.

Consistent Processing

Regardless of the input format, your downstream analysis and transformation logic should work identically.

Swappable Components

You want to change one component (like switching from CSV to JSON) without rewriting your entire pipeline.

Our goal: **build the same pipeline once, but support different input and output implementations seamlessly**. This is where inheritance, polymorphism, and interfaces become invaluable.

CORE CONCEPT

Inheritance: What It Is

Inheritance is one of the fundamental pillars of object-oriented programming. It allows you to create a new class that builds upon an existing class, inheriting its attributes and methods whilst adding or modifying functionality.

The key principle: **a subclass *is a* specialised version of a base class**. This "is-a" relationship is crucial—inheritance should represent a genuine subtype, not just a convenient way to share code.

Common Pattern

```
class BaseLoader:
    # Shared functionality here
    ...

class CSVLoader(BaseLoader):
    # CSV-specific implementation
    ...
```

When to Use Inheritance

- You need a shared contract across multiple implementations
- Common behaviour can be defined once in the base class
- Multiple implementations of the same "role" or responsibility
- A clear "is-a" relationship exists

Polymorphism: The Big Payoff

Polymorphism is where object-oriented design truly shines. The term literally means "many forms"—it allows your code to work with objects through their base type whilst the actual behaviour depends on the specific subclass being used.

The Power of Polymorphism

Write your code once against the base type, and it automatically works with any subclass. This means you can swap implementations without changing the code that uses them.

```
loader: BaseLoader = CSVLoader()
dataset = SalesDataset(loader=loader)
dataset.load()

# Later, switch to JSON...
loader: BaseLoader = JSONLoader()
# Same code still works!
```

Real Benefits

- **Flexibility:** Change implementations without touching client code
- **Testability:** Easily swap in mock objects for testing
- **Maintainability:** Add new implementations without modifying existing code
- **Scalability:** Extend systems without breaking existing functionality

Interfaces & Contracts with ABC

An interface is a formal promise: "these methods will exist". Python uses Abstract Base Classes (ABC) to define interfaces that subclasses must implement. This contract ensures consistency across implementations and catches errors early.

Defining an Interface

```
from abc import ABC, abstractmethod

class BaseLoader(ABC):
    @abstractmethod
    def load(self, path: str):
        """Load data from the given path.

        Must be implemented by all subclasses.
        """
        pass

    @abstractmethod
    def validate(self) -> bool:
        """Validate loaded data."""
        pass
```

Key Benefits

- **Consistent APIs**

  All implementations must provide the same methods with the same signatures.

- **Early Error Detection**

  Python raises TypeError if you try to instantiate a class that hasn't implemented all abstract methods.

- **Replaceable Components**

  Guaranteed that any subclass can substitute for the base class.

ADVANCED TOPIC

Alternative Contract: Protocol

Python's `Protocol` from the `typing` module offers an alternative approach to defining interfaces—one based on **structural typing** rather than inheritance. This is sometimes called "duck typing" formalised.

How Protocol Differs

With Protocol, a class doesn't need to explicitly inherit from an interface. If it has the right methods with the right signatures, it automatically satisfies the protocol. This is useful when you don't want to force inheritance or when working with third-party classes.

```
from typing import Protocol

class Loadable(Protocol):
    def load(self, path: str): ...
    def validate(self) -> bool: ...
```

When to Consider Protocol

- You don't control the class definitions
- You want to avoid inheritance hierarchies
- You're defining interfaces for type checking only
- You need flexibility with third-party code

You don't have to use Protocol today, but it's valuable to know it exists as an alternative to ABC for more advanced scenarios.

Composition vs Inheritance

Choosing the Right Relationship

One of the most important design decisions you'll make is choosing between inheritance and composition. Both are powerful tools, but they serve different purposes and have different implications for your code's flexibility and maintainability.

Inheritance: "Is-A"

Represents a specialisation relationship. A CSVLoader **is a** BaseLoader. Use when the subclass is genuinely a type of the base class.

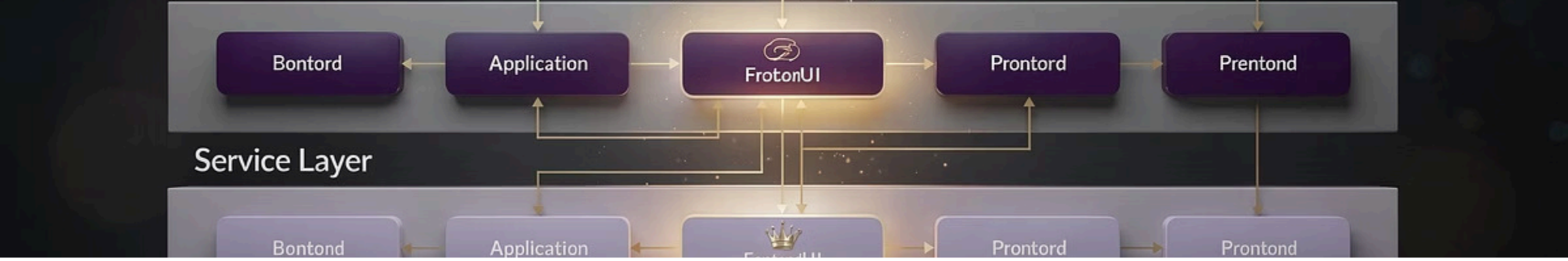
- Shares interface and behaviour
- Creates tight coupling
- Good for polymorphism

Composition: "Has-A"

Represents a usage relationship. A SalesDataset **has a** loader. Use when you want to use functionality without being that type.

- Flexibility to swap components
- Loose coupling
- Easier to test and modify

Rule of thumb for data pipelines: Composition is often cleaner and more flexible. A SalesDataset has a loader; an Analyser uses a dataset. This keeps components independent and reusable.



Example Architecture

A Well-Designed Data Pipeline

Let's examine how these OOP principles come together in a practical data pipeline architecture. This design separates concerns cleanly, making the system flexible, testable, and maintainable.

BaseLoader

Abstract interface defining the contract for all data loaders

Concrete Loaders

CSVLoader, JSONLoader, APILoader—each implementing BaseLoader

SalesDataset

State holder with path and rows; uses a loader via composition

SalesAnalyser

Pure logic for KPIs and metrics; uses a dataset

Key insight: You can swap the loader implementation without changing a single line in the analyser. The SalesDataset doesn't care whether it's loading from CSV, JSON, or an API—it just calls the loader's interface. This is the power of polymorphism and composition working together.

PROPERTIES

@property: Computed Attributes

Properties are a Pythonic way to expose computed values that look like simple attributes but actually run code when accessed. They provide a clean, intuitive API whilst maintaining control over how values are calculated.

Common Use Cases

- **Computed values:** Calculate total_revenue from quantity × price
- **Read-only views:** Expose data without allowing modification
- **Validation on access:** Check state before returning values
- **Lazy loading:** Compute expensive values only when needed

```
class Sale:
 def __init__(self, quantity: int, unit_price: float):
 self.quantity = quantity
 self.unit_price = unit_price

 @property
 def revenue(self) -> float:
 """Computed revenue property."""
 return self.quantity * self.unit_price

# Usage: clean and intuitive
sale = Sale(quantity=10, unit_price=99.99)
print(sale.revenue) # Looks like an attribute!
```

Why Properties Are Useful

Properties offer significant advantages for building maintainable, Pythonic code. They strike an elegant balance between simplicity and control, allowing you to change internal implementation without breaking external code.

1

Simple, Clean API

Access computed values with simple attribute syntax (`obj.total_revenue`) instead of method calls (`obj.total_revenue()`). This makes your classes feel more natural to use.

2

Future-Proof Implementation

Start with a simple attribute, then later change it to a property with computation—without changing any code that uses it. This is incredibly valuable for evolving codebases.

3

Enforce Read-Only Access

Create properties without setters to prevent external modification. This helps maintain object invariants and prevents bugs from unexpected state changes.

Important caution: Avoid performing heavy computations inside properties unless the results are cached. Users expect attribute access to be fast. If computation is expensive, consider using an explicit method or caching the result.

Dunder Methods: Why They Matter

Dunder methods (short for "double underscore") are special methods that integrate your custom classes with Python's built-in language features. They're also called "magic methods", and they allow your objects to behave like native Python types.

Essential Dunder Methods for Data Projects

__repr__ / __str__

Readable string representations for debugging and logging

__len__

Enable len(dataset) to return the number of items

__iter__

Allow iteration: for row in dataset

__getitem__

Support indexing: dataset[0] or dataset["column"]

The Benefits

Implementing dunder methods makes your classes feel like native Python objects. Users can apply familiar operations and syntax without learning a custom API. This significantly improves usability and reduces cognitive load.

For data structures especially, dunder methods are essential. They allow your custom dataset or collection classes to work seamlessly with Python's iteration protocols, indexing syntax, and built-in functions.

__repr__: Your Debugging Superpower

The `__repr__` method returns a string representation of your object that's primarily intended for developers. A well-crafted `__repr__` is invaluable during debugging, logging, and interactive development sessions.

Implementation Example

```
def __repr__(self) -> str:
    return (
        f"SalesDataset("
        f"path={self.path!r}, "
        f"n_rows={len(self)}"
        f")"
    )

# In the interpreter or logs:
>>> dataset
SalesDataset(path='sales_2024.csv', n_rows=1523)
```

The `!r` formatting ensures string values are quoted, making the output more precise and unambiguous.

What Makes a Good __repr__

- One line, concise but informative
- Shows the class name clearly
- Includes key identifying information
- Uses repr() for string attributes (via !r)
- Ideally, could recreate the object

Goal: Create a one-line summary you can trust when debugging at 2am.

__len__ and __iter__

Make Your Objects Collection-Like

Two of the most useful dunder methods for data structures are `__len__` and `__iter__`. Together, they allow your custom objects to behave like built-in Python collections, making them intuitive and Pythonic to use.

__len__: Count Your Items

```
def __len__(self) -> int:
"""Return the number of rows in the dataset."""
return len(self.rows)


# Now you can use the built-in len() function:
print(f"Dataset contains {len(dataset)} rows")
```

This simple method unlocks the `len()` built-in function for your objects. It makes your dataset immediately understandable—users instinctively know how to check its size.

__iter__: Enable Iteration

```
def __iter__(self):
    """Make the dataset iterable."""
    return iter(self.rows)

# Now your dataset works with for loops:
for row in dataset:
    process(row)
```

The `__iter__` method allows your object to work with for loops, list comprehensions, and any other iteration context. This is fundamental for data structures.

With these two methods implemented, your dataset behaves like a proper Python collection—and users can apply all their existing knowledge of lists and sequences.

__getitem__: Access Patterns

The `__getitem__` method enables square bracket notation for accessing items in your custom objects. However, you need to make an important design choice: what should the brackets mean for your specific class?

Index-Based Access

Treat your object like a list, where `dataset[0]` returns the first row, `dataset[1]` returns the second row, and so on. This is intuitive for row-based data structures.

```
def __getitem__(self, index: int):
    return self.rows[index]

# Usage
first_sale = dataset[0]
```

Column-Based Access

Treat your object like a dictionary of columns, where `dataset["product"]` returns a list of all product values. This is convenient for column-oriented analysis.

```
def __getitem__(self, column: str):
    return [row[column] for row in self.rows]

# Usage
products = dataset["product"]
```

Important: Choose one pattern and stick with it, or support both but document the behaviour clearly. Inconsistent behaviour confuses users and leads to bugs.

⚠ WARNING

When Inheritance Goes Wrong

Whilst inheritance is powerful, it's also easy to misuse. Poor inheritance design creates brittle, hard-to-maintain code. Understanding common anti-patterns helps you avoid these pitfalls and build better systems.

Deep Inheritance Trees

Avoid creating inheritance hierarchies more than 2-3 levels deep. Deep trees become difficult to understand, modify, and debug. Each level adds coupling and complexity.

"God Base Class" Problem

Don't create a base class that tries to do everything, with dozens of methods and unrelated responsibilities. This violates the Single Responsibility Principle and creates tight coupling.

Inheritance for Code Sharing Only

Don't use inheritance merely to avoid duplicating code when there's no genuine "is-a" relationship. Use composition, helper functions, or mixins instead.

The Golden Rule

Inheritance should express a **real, meaningful subtype relationship**.
Ask yourself: "Is this genuinely a specialised version of the base class?"
If you're hesitant, composition is probably better.

Warning Signs

- You're overriding most base class methods
- The subclass doesn't use base class functionality
- You need to disable inherited behaviour
- The relationship feels forced or unnatural

Summary: What You Can Now Build

You now possess the tools to design sophisticated, professional-grade object-oriented systems. These aren't just academic concepts—they're the building blocks of maintainable, scalable data applications used in production environments worldwide.

Replaceable Components

Use polymorphism and contracts (ABC) to build systems where components can be swapped without rewriting client code. Change data sources, processing logic, or output formats independently.

Clear Separation of Responsibilities

Apply composition to keep concerns separated. Each class has a single, well-defined purpose. Changes to one component don't cascade through the system.

Strong, Pythonic Objects

Use properties and dunder methods to create objects that feel native to Python. Your classes integrate seamlessly with built-in functions, iteration, and standard syntax.

Mental Checklist

Before Writing OOP Code

Before you start implementing a class-based solution, pause and work through these fundamental questions. This checklist helps you make deliberate design decisions rather than falling into coding patterns by habit.

01

What is the "thing"?

Identify the entity and its essential state. What data must this object hold? What makes one instance different from another?

02

What behaviors belong to it?

Determine which operations truly belong to this class. What can this object do? What questions can you ask it? Avoid adding methods that don't relate to the core responsibility.

03

Do I need multiple implementations?

Consider whether you'll have different versions of this role. If yes, you need polymorphism via inheritance or interfaces. If no, keep it simple.

04

Is this "is-a" or "has-a"?

Determine the relationship type. Is this a specialisation (inheritance) or a usage (composition)? When in doubt, prefer composition—it's more flexible.

05

What's the simplest API?

Think from a teammate's perspective. What would be the most intuitive way to use this class? Favour clarity and simplicity over cleverness.