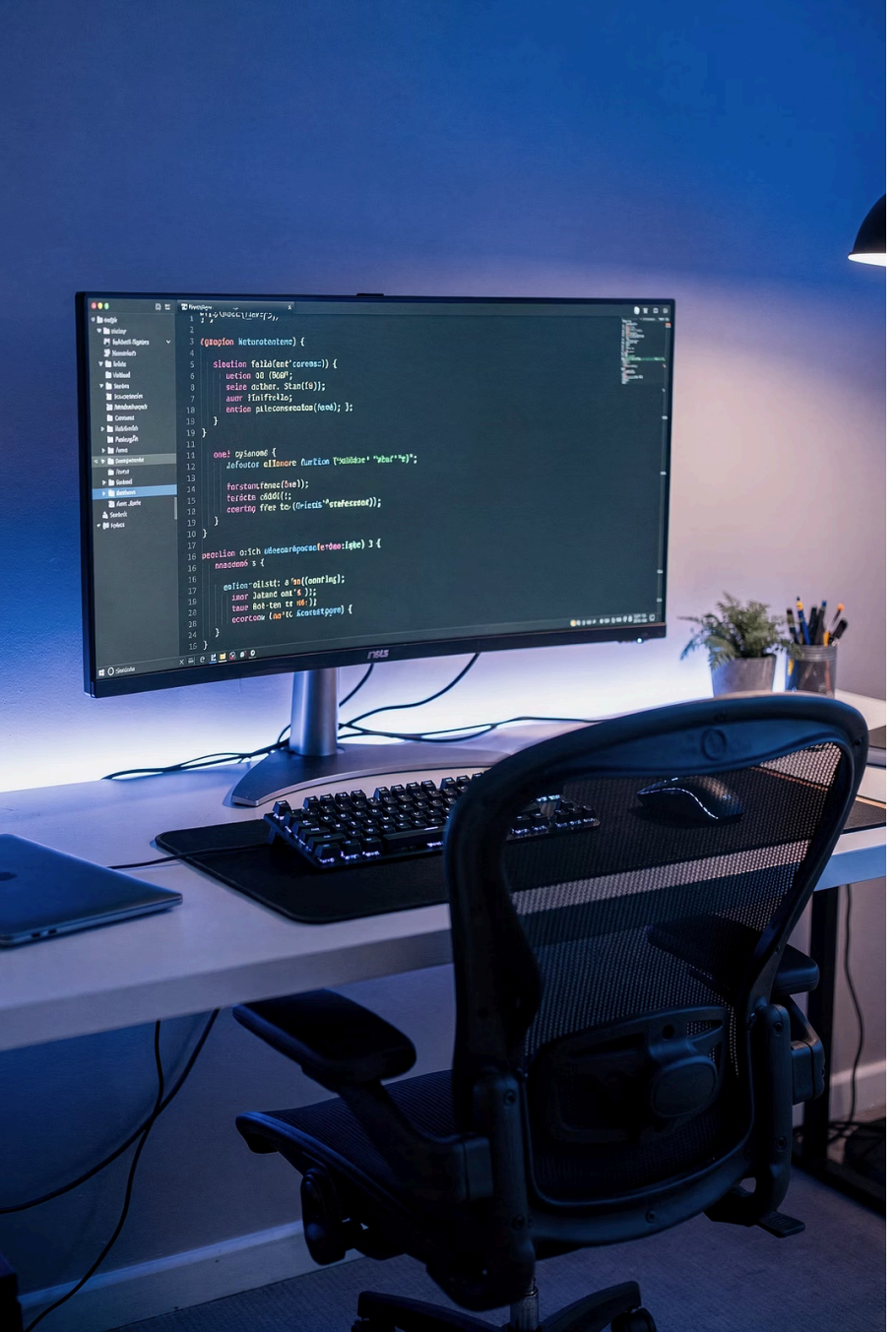


Session 2: Object-Oriented Programming (OOP) Basics for Data Projects

A practical introduction to structuring your data workflows with classes and objects

LEARNING OBJECTIVES

What You'll Master Today

By the end of this session, you'll have a solid foundation in object-oriented programming principles specifically tailored for data work. These skills will transform how you organise and manage your data projects.

1

Classes and Objects

Understand the distinction between a class blueprint and object instances, and when to use each in your data workflows

2

Core Mechanics

Master the use of self, __init__, attributes, and methods to build functional data structures

3

API Design

Design clean, intuitive object interfaces that are easy to use, test, and maintain over time

4

Real-World Application

Apply OOP principles to structure complete data workflows: loading, analysing, and exporting data

Why Use OOP in Data Work?

Object-oriented programming becomes invaluable when you're working with concepts that you manipulate repeatedly throughout your project. Rather than scattering related functions and variables across multiple files, OOP lets you bundle everything together logically.

Consider these common scenarios in data projects:

- A **dataset** that needs loading, cleaning, and analysis
- A **report generator** with specific formatting requirements
- A **data pipeline** with multiple processing stages

Each of these involves both **internal state** (like file paths, configuration settings, or cached data) and **operations** that work with that state (loading, computing metrics, exporting results).

The Core Principle

Group data + behaviour into one coherent unit. This approach makes your code more intuitive, easier to test, and simpler to maintain as your project grows.

Essential OOP Vocabulary

Before we dive into practical examples, let's establish a shared vocabulary. These five concepts form the foundation of object-oriented programming.

Class

The blueprint or template that defines the structure and capabilities of objects. Think of it as the architectural plan before construction.

Object / Instance

A concrete thing created from a class blueprint. Each instance can have different data whilst sharing the same structure.

Attributes

The stored state—the data living inside an object. These are the variables that remember information between method calls.

Methods

Functions that operate on the object's data. They define what actions your object can perform.

self

A reference to the current object instance. It's how methods access and modify the object's own attributes.

Class vs Instance: A Simple Example

Let's see these concepts in action with a straightforward Counter class. This example demonstrates how multiple instances can exist independently, each maintaining its own state.

```
class Counter:
    def __init__(self):
        self.value = 0

    def inc(self):
        self.value += 1

c1 = Counter()
c2 = Counter()
c1.inc()
print(c1.value)  # 1
print(c2.value)  # 0
```

What's Happening Here?

- **Counter** is the class—the blueprint defining how counters work
- **c1** and **c2** are separate instances created from that blueprint
- Each instance has its own **value** attribute that changes independently
- Calling **inc()** on c1 doesn't affect c2

Key Takeaway: Each instance maintains its own state. Changes to one object don't affect other objects created from the same class.

The __init__ Method: Setting Up Your Object

The `__init__` method is Python's constructor—it runs automatically when you create a new instance. Its primary job is to initialise the minimum required state for your object to function properly.

Best Practices

- **Keep it simple**

  Don't perform heavy operations like file I/O or network requests here

- **Store essential inputs**

  Save the parameters you'll need later as attributes

- **Set safe defaults**

  Initialise containers like lists and dictionaries to empty states

Example: A Data Project Class

```
class SalesDataset:
    def __init__(self, path: str):
        self.path = path
        self.rows = []  # filled later
        self._loaded = False
```

Notice how `__init__` only *stores* the file path—it doesn't actually read the file. The heavy lifting happens later when you explicitly call a `load()` method.

Attributes vs Methods: Data vs Actions

Understanding the distinction between attributes and methods is crucial for designing clean, intuitive classes. Think of attributes as the "nouns" of your object and methods as the "verbs".

Attributes

Stored values that represent the object's current state. These are typically set in __init__ or modified by methods.

Methods

Actions that use or modify the object's state. They implement the behaviour and capabilities of your class.

Practical Example Pattern

```
class SalesDataset:
    def load(self) -> None:
        """Load data from self.path into self.rows"""
        # Read CSV file and populate self.rows
        with open(self.path) as f:
            self.rows = list(csv.DictReader(f))
        self._loaded = True

    def total_revenue(self) -> float:
        """Calculate total revenue from loaded rows"""
        if not self._loaded:
            raise ValueError("Data not loaded. Call load() first.")
        return sum(
            float(r["quantity"]) * float(r["unit_price"])
            for r in self.rows
        )
```

Notice how `load()` modifies the object's state whilst `total_revenue()` uses that state to perform a calculation. This separation of concerns makes the code easier to understand and test.

Object Lifecycle: The Journey from Creation to Results

Well-designed objects follow a predictable lifecycle. Understanding this pattern helps you structure your classes logically and makes them more intuitive to use.

01

Create

Instantiate the object with necessary configuration (file paths, settings, parameters)

02

Load

Build or load the internal state (read files, fetch data, initialise structures)

03

Compute

Perform calculations and transformations on the loaded data

04

Export

Output results (save to file, display to screen, return processed data)

Seeing It in Action

```
ds = SalesDataset("data/sales.csv")
ds.load()
revenue = ds.total_revenue()
print(f"Total revenue: £{revenue:,.2f}")
ds.export_summary("report.json")
```

This clear sequence makes the code self-documenting. Anyone reading it immediately understands: create the dataset object, load the data, compute metrics, and export results. No surprises, no hidden steps.

Designing a Clean Object API

Your object's API—the methods and attributes users interact with—should be obvious and concise. A well-designed API reduces cognitive load and makes your code a pleasure to work with.

Predictable Method Names

Use clear, conventional names like `load()`, `save()`, `calculate_total()`. Avoid cryptic abbreviations or unusual terminology.

Safe Defaults

Methods should handle edge cases gracefully. An empty dataset should return 0 for totals, not crash. Missing optional parameters should use sensible defaults.

Clear Contracts

Document what must happen before a method is called. If `total_revenue()` requires calling `load()` first, make this clear—ideally with helpful error messages.

Minimal Complexity

Expose only what users need. Internal helper methods can remain private (prefixed with underscore). Keep the public interface small and focused.

Minimal Encapsulation for Beginners

Encapsulation means the object controls how its internal state changes. You don't need complex getters and setters, just prefer method calls over direct attribute manipulation when it matters.

The Principle

Interact with objects through their methods whenever those methods provide meaningful behaviour or validation. This prevents accidental inconsistencies in the object's state.

✅ Preferred Approach

revenue = ds.total_revenue()

ds.add_transaction(qty, price)

Methods encapsulate logic and maintain consistency

⚠️ Risky Practice

ds.rows.append(some_dict)

revenue = sum(...) # outside

Direct manipulation risks breaking internal invariants

Balance is key: Don't over-engineer with excessive encapsulation, but do use methods for operations that have business logic or need validation.

Debugging Comfort: Making Objects Readable

When debugging, you'll frequently print objects to inspect their state. By default, Python shows cryptic memory addresses. Implementing `__repr__` or `__str__` makes your debugging sessions far more productive.

Before: Unhelpful Output

```
>>> ds = SalesDataset("data.csv")
>>> print(ds)
<__main__.SalesDataset object at 0x7f8b3c4d5e80>
```

This tells you almost nothing about the object's actual state.

After: Informative Output

```
>>> ds = SalesDataset("data.csv")
>>> print(ds)
SalesDataset(path='data.csv', n_rows=1247)
```

Now you instantly see what data the object contains.

The Implementation

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

    def __str__(self) -> str:
        return f"Sales data from {self.path} ({len(self.rows)} rows)"
```

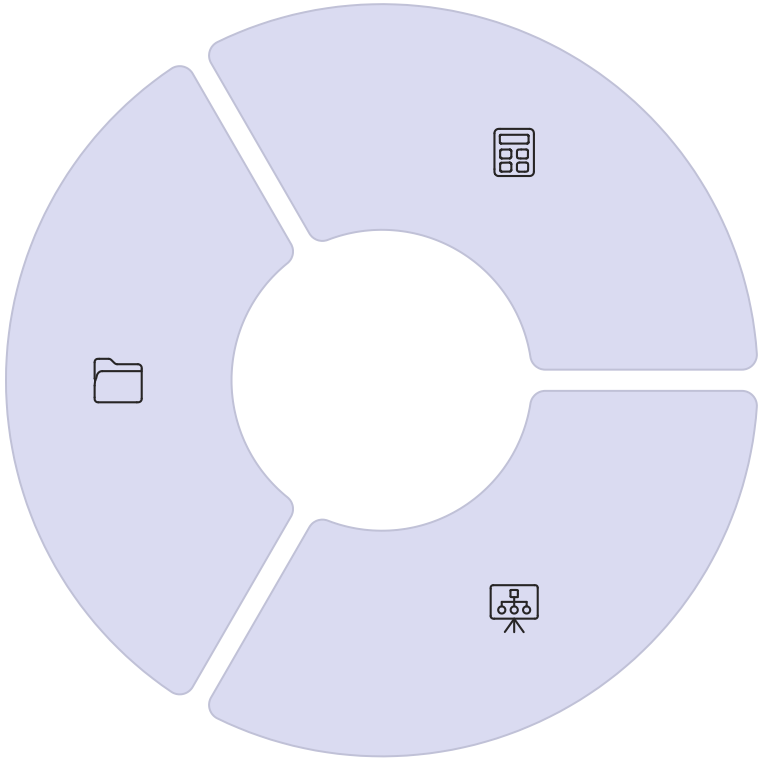
`__repr__` should be unambiguous and developer-focused, whilst `__str__` can be more user-friendly. When in doubt, implement `__repr__` first.

OOP + Clean Separation of Responsibilities

Object-oriented programming doesn't replace good design principles—it amplifies them. Even with classes and objects, you still need clear separation of concerns.

I/O Layer

File reading, writing, parsing, and data serialisation. Keep these operations isolated.



Business Logic

Calculations, transformations, validations, and domain-specific rules. The core intelligence.

Orchestration

The main script that coordinates objects and calls methods in the right sequence.

Consider this structure: your `SalesDataset` class focuses on representing sales data and computing metrics. A separate `CSVLoader` utility handles file parsing. Your `main()` function orchestrates the workflow. This separation makes each component testable in isolation and easier to modify.

Common OOP Mistakes to Avoid

Learning from common pitfalls will save you hours of frustration. Here are the mistakes beginners make most frequently when starting with object-oriented programming.

The "God Object" Anti-Pattern

Creating one giant class that does everything: loads files, processes data, generates reports, sends emails, and makes coffee. This defeats the purpose of OOP. **Solution:** Split responsibilities into multiple focused classes.

Heavy __init__ Methods

Performing file I/O, network requests, or complex calculations in `__init__`. This makes object creation slow and error-prone. **Solution:** Use `__init__` only for simple initialisation; add separate `load()` or `build()` methods for heavy work.

Methods That Do Too Much

A single 200-line method that reads data, cleans it, computes statistics, and generates a report. Impossible to test or reuse. **Solution:** Break large methods into smaller, single-purpose methods.

Hidden Side Effects

A method called `get_summary()` that also saves a file to disk. Users expect getters to be read-only. **Solution:** Name methods clearly to indicate when they modify state or perform I/O: `generate_and_save_summary()`.

SUMMARY

Key Concepts Recap

You've learnt the fundamentals of object-oriented programming specifically for data projects. Let's consolidate the essential points you'll use in your own code.

Classes & Objects

Classes define the structure and behaviour; objects are concrete instances that hold actual data and maintain independent state.

Core Mechanics

`__init__` initialises the minimum required state. Methods implement actions. Attributes store data between method calls.

API Design

A clean API with predictable names, safe defaults, and clear contracts makes your code easier to reuse, test, and maintain over time.

Design Principles

- Follow a clear object lifecycle: create, load, compute, export
- Prefer method calls over direct attribute manipulation
- Implement `__repr__` for better debugging
- Separate I/O, logic, and orchestration concerns

Common Pitfalls

- Avoid giant "god objects" that do everything
- Keep `__init__` light—no heavy I/O
- Break long methods into focused functions
- Name methods to reveal side effects clearly