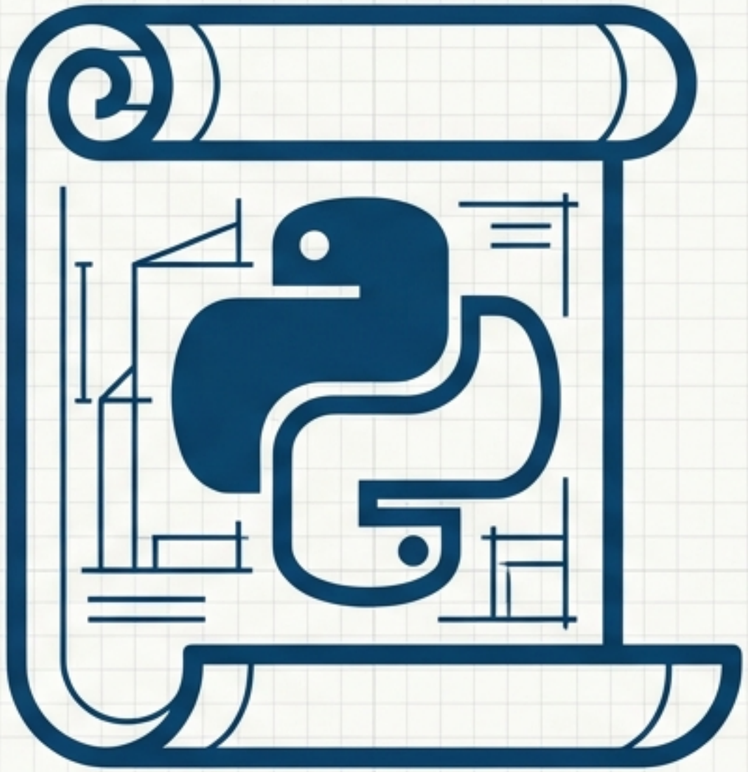


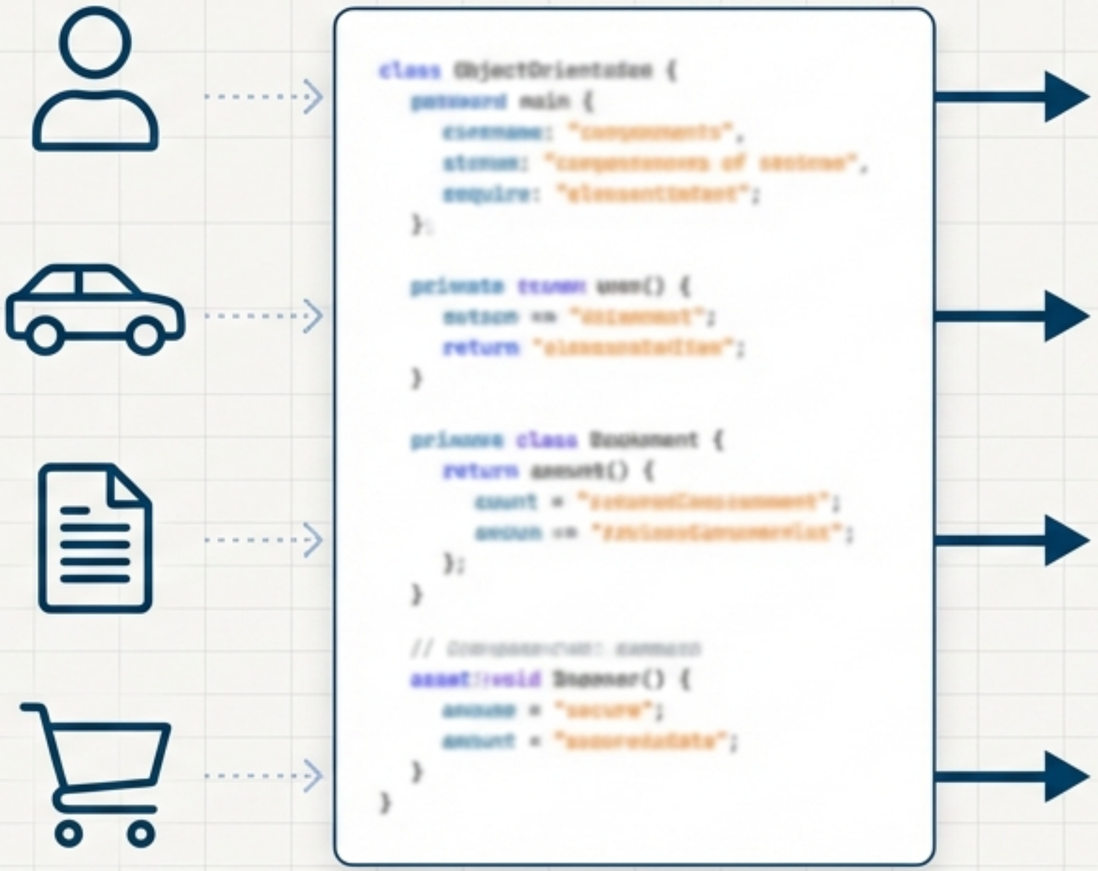
From Blueprint to Application: A Guide to Python OOP

A structured journey into building robust, scalable, and real-world software with Object-Oriented principles.

OOP exists to model the real world in code.

The primary goal of Object-Oriented Programming is to manage complexity by creating models of real-world entities.

It provides a way to structure code that is:



- **Reusable**: Build components that can be used in multiple projects.
- **Modular**: Isolate parts of a system to work on them independently.
- **Scalable**: Design systems that can grow without becoming chaotic.
- **Secure**: Protect data and hide implementation details from the outside world.

The `Class` is the blueprint for creating objects.

Concept

- A `Class` is a template that defines the properties (attributes) and behaviours (methods) that all objects of that type will have.
- It doesn't represent a concrete thing, but the *idea* of a thing.

Core Syntax

```
class ClassName:
    # class attributes and methods go here
    pass
```

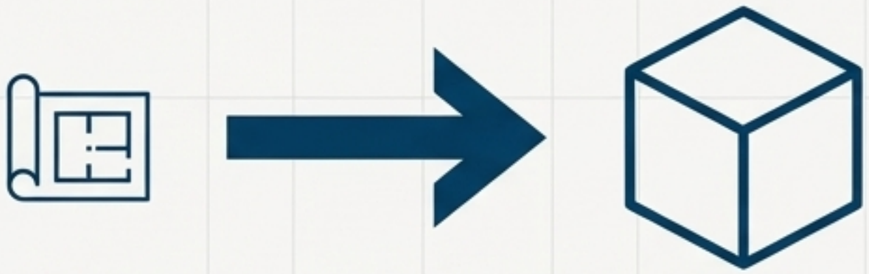
Simple Example: A blueprint for a 'Student'.

```
class Student:
    name = "Default" # A class variable
```

An `Object` is a concrete instance of a class.

Concept

If a `Class` is the blueprint for a house, an `Object` is an actual house built from that blueprint.



You can create many unique objects from a single class, each with its own state.

Syntax & Example

Creating an Instance (Instantiation)

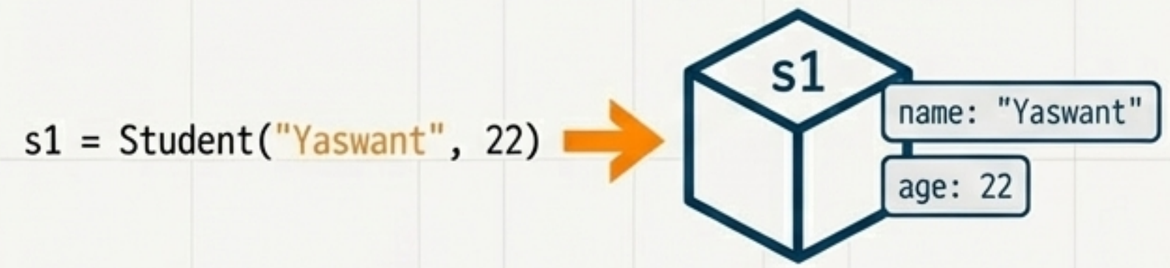
```
# s1 is an object, or an instance of the Student class
s1 = Student()
```

Accessing Attributes

```
# Access the 'name' attribute defined in the class
print(s1.name)
# Output: Default
```

The `__init__()` constructor initialises each new object.

Known as a constructor, the `__init__` method is known as a constructor, the `__init__` method is called automatically when you create a new object. Its job is to set up the object's initial state, typically by assigning values to its instance variables. The `self` parameter refers to the newly created instance.




```python
class Student:
    # The constructor for the class
    def __init__(self, name, age):
        # These are 'instance variables'
        self.name = name
        self.age = age

# Now, we provide initial values when creating the object
s1 = Student("Yaswant", 22)

print(s1.name) # Output: Yaswant
print(s1.age)  # Output: 22
```

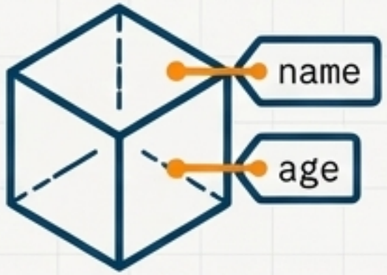
Objects hold their own data and know how to act on it.

Instance Variables

Data that is specific to each object instance. They are defined inside `__init__` using the `self` keyword (e.g., `self.name`).

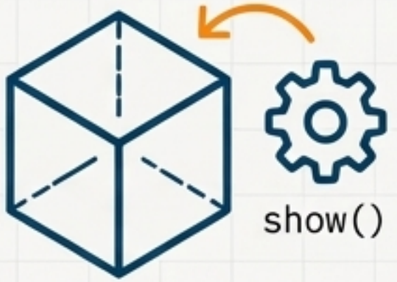
Purpose

To maintain the unique state of each object.



Instance Methods

Functions defined inside a class that operate on an object's instance variables. They always take `self` as their first argument.




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

    # This is an instance method
    def show(self):
        print(f"Name: {self.name}, Age: {self.age}")

s1 = Student("Yaswant", 22)
s1.show() # Calling the instance method
# Output: Name: Yaswant, Age: 22
```

Some features belong to the blueprint, not the building.

Class Variables & Methods

Variables: Shared by all instances of the class.

Methods: Decorated with `@classmethod`, they receive the class (`cls`) as the first argument. Used to modify class state.

```
class Student:
    school = "ABC School"  # Class variable

    @classmethod
    def change_school(cls, new_name):
        cls.school = new_name

Student.change_school("XYZ School")
```

Static Methods

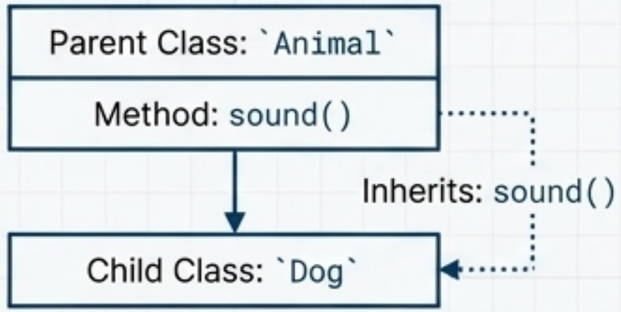
Definition: Decorated with `@staticmethod`. They don't receive `self` or `cls`. Essentially a normal function namespaced within the class.

Purpose: Used for utility functions that are logically related to the class but don't depend on any class or instance state.

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b

result = Math.add(5, 3) # Call directly on the class
```

Inheritance lets you build on proven designs.

Inheritance is a mechanism where a new class (the child or subclass) acquires the properties and methods of an existing class (the parent or superclass). This promotes an 'is-a' relationship (e.g., a `Dog` *is an* `Animal`).



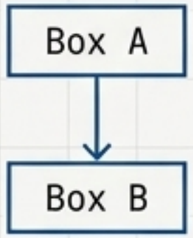
Benefits
****Code Reusability****: Avoids duplicating code.
****Logical Structure****: Creates a clear and hierarchical relationship between classes.

```
# Parent Class
class Animal:
    def sound(self):
        print("This animal makes a sound.")

# Child Class inherits from Animal
class Dog(Animal):
    pass # It inherits 'sound' without any extra code

d = Dog()
d.sound() # We can call the parent's method on the child object
# Output: This animal makes a sound.
```

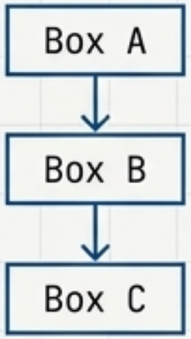
Managing family lines: patterns and tools.

Types of Inheritance

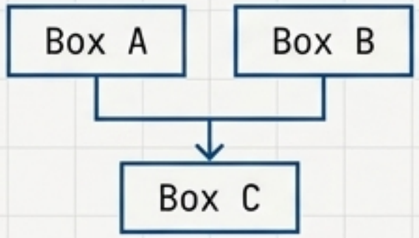
Python supports several patterns for combining blueprints:



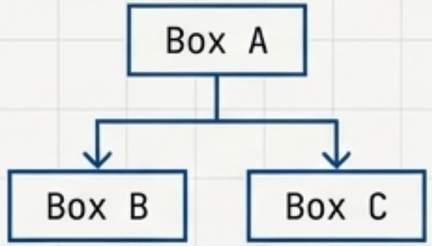
Single



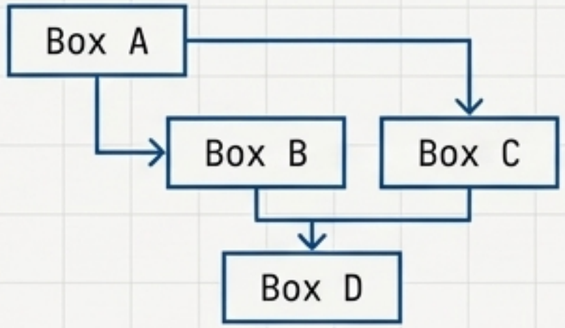
Multilevel



Multiple



Hierarchical



Hybrid

The `super()` Function

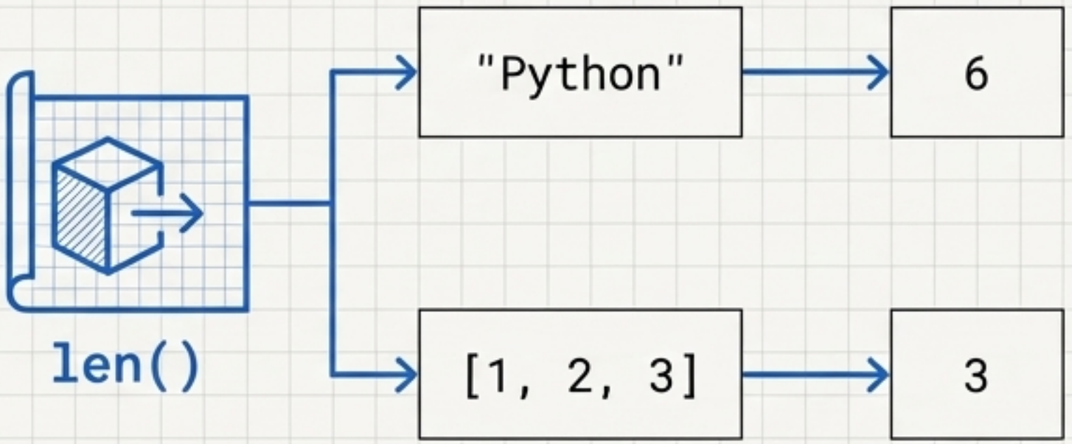
The `super()` function allows a child class to call methods from its parent class. It is essential for extending, not just replacing, parent functionality.

```
class A:
    def show(self):
        print("In A's show method")

class B(A):
    def show(self):
        # Call the parent's 'show' method first
        super().show()
        print("In B's show method")

b = B()
b.show()
# Output:
# In A's show method
# In B's show method
```

Polymorphism means “one name, many forms”.

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It's the ability for a single method name to have different behaviours depending on the object it's called on.

Built-in Function Polymorphism



The `len()` function behaves differently depending on the object type.

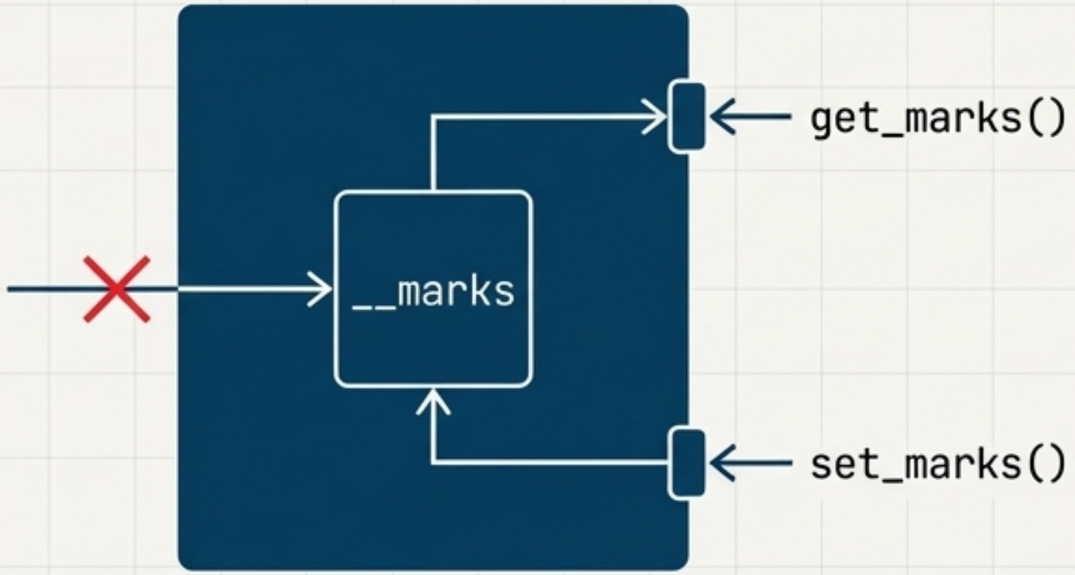
Method Overriding in Classes

A child class can provide a specific implementation of a method that is already provided by its parent class.

```
class A:
    def show(self):
        print("Show from A")

class B(A):
    def show(self): # Overriding the parent's method
        print("Show from B")
```

Encapsulation creates a secure black box for your data.

Encapsulation is the practice of bundling data (attributes) and the methods that operate on that data within a single unit (the class). It also involves restricting direct access to an object's components.



Private Variables: Use a double underscore prefix (`__`) to make an attribute 'private'. Python will 'mangle' the name to make it difficult to access from outside the class.

Controlled Access: Public 'getter' and 'setter' methods are used to safely access or modify private variables, allowing for validation.

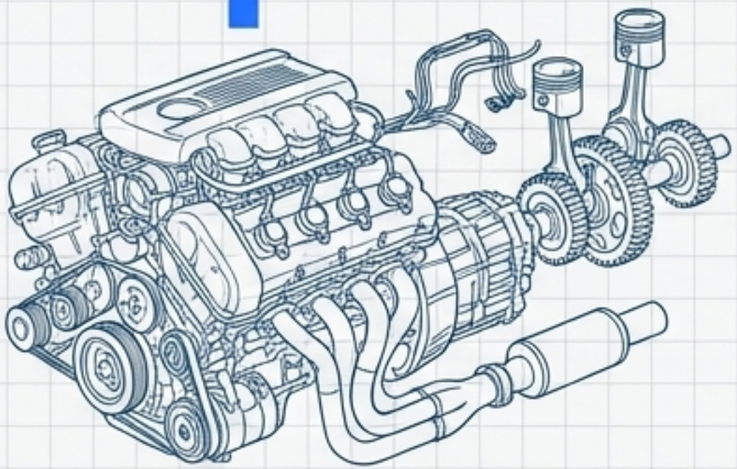
```
class Student:
    def __init__(self):
        self.__marks = 0 # Private variable

    def get_marks(self): # Getter
        return self.__marks

    def set_marks(self, value): # Setter with validation
        if 0 <= value <= 100:
            self.__marks = value
```

Abstraction shows only what's necessary.

Concept & Diagram

Abstraction focuses on exposing an object's essential features while hiding the irrelevant or complex implementation details. It defines a 'contract' that other classes must follow.

User Interface
(The Abstraction)



Implementation Details
(Hidden)

Implementation & Example

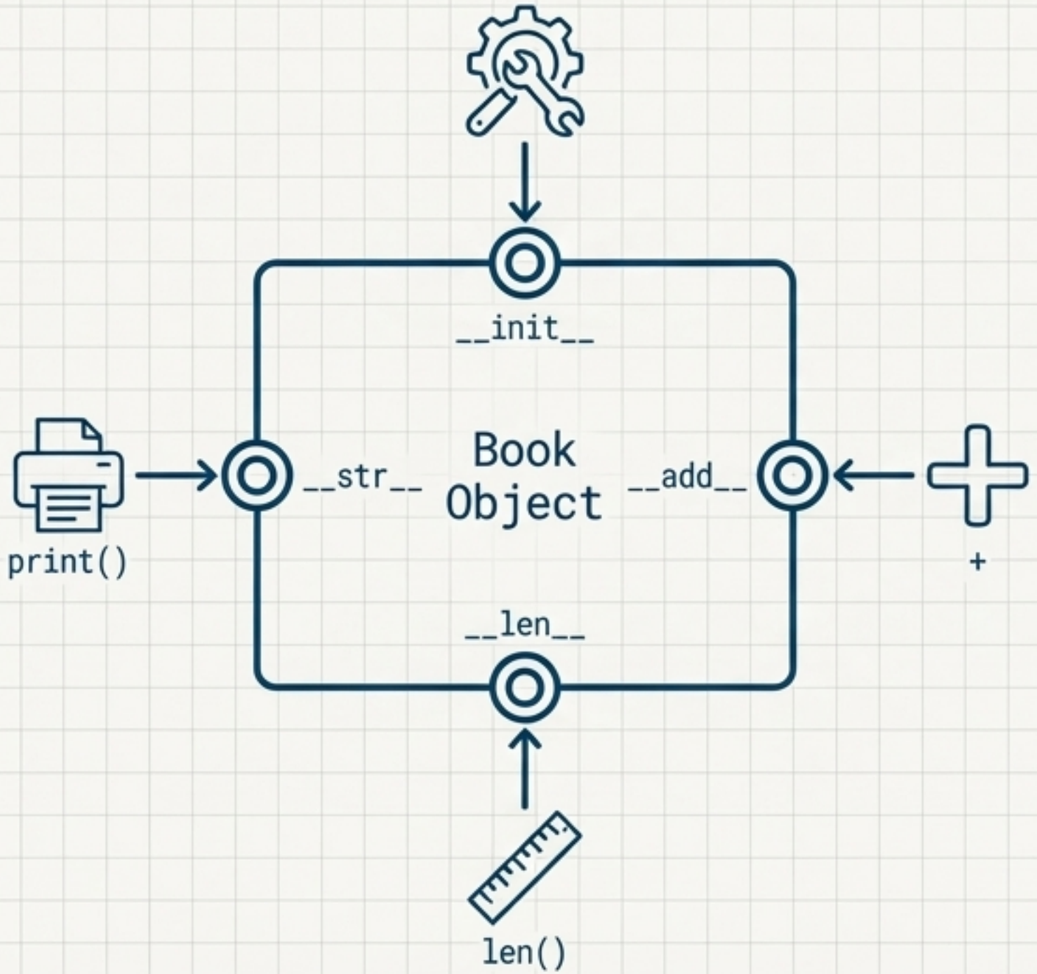
Python's `abc` module (Abstract Base Classes) is used to create abstract classes and methods.

An abstract method is declared but contains no implementation. Subclasses are forced to implement it.

```
from abc import ABC, abstractmethod

# An abstract class that defines a contract
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass # Subclasses MUST implement this method

class Square(Shape):
    def area(self): # Fulfilling the contract
        # ... implementation for square area
        return ...
```

Magic methods let your objects speak Python's native language.

Dunder (double underscore) methods are predefined methods in Python that you can override to change the default behaviour of your objects. They allow your objects to work with built-in operators and functions.



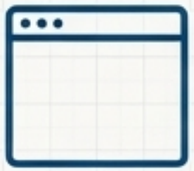
Common Examples

- `__init__(self, ...)`: Object initialisation.
- `__str__(self)`: Called by `str()` and `print()`. Should return a user-friendly string.
- `__len__(self)`: Called by `len()`. Should return the object's length.
- `__add__(self, other)`: Used for the + operator.
- `__eq__(self, other)`: Used for the == equality operator.

```
class Book:
    def __init__(self, title):
        self.title = title

    def __str__(self):
        return f"A book titled '{self.title}'"

my_book = Book("Python OOP")
print(my_book) # __str__ is called automatically
# Output: A book titled 'Python OOP'
```

These concepts are the foundation of modern software.

Object-Oriented Programming is not just an academic exercise. It is the dominant paradigm used to build complex, real-world applications across every field of software development.



Web Applications

Modelling `User`, `Product`, and `Order` objects in e-commerce sites.

Game Development

Creating `Character`, `Enemy`, and `Item` classes with unique attributes and behaviours.

AI & Machine Learning

Building modular data processing `Pipelines` and `Model` objects.

Cybersecurity

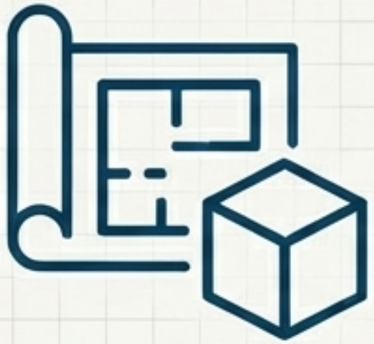
Designing `Scanner` or `Packet` objects to analyse network traffic.

General Modelling

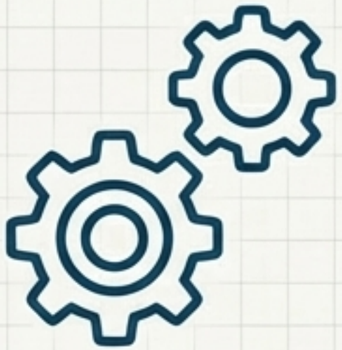
Representing any real-world entity, from `Employee` and `Student` records to `Car` and `Device` simulations.

Your journey: from blueprint to organised systems.

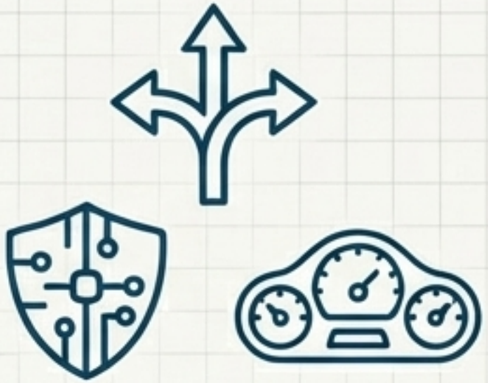
You have progressed from the fundamental building blocks to the high-level principles that guide robust software architecture.



Classes & Objects: The core relationship of blueprint and instance.



Inheritance: The tool for reusing and extending code.



The Pillars: The principles for creating flexible, secure, and simple interfaces.

Mastering OOP means moving beyond writing scripts to engineering systems. It is the language of building software that lasts.