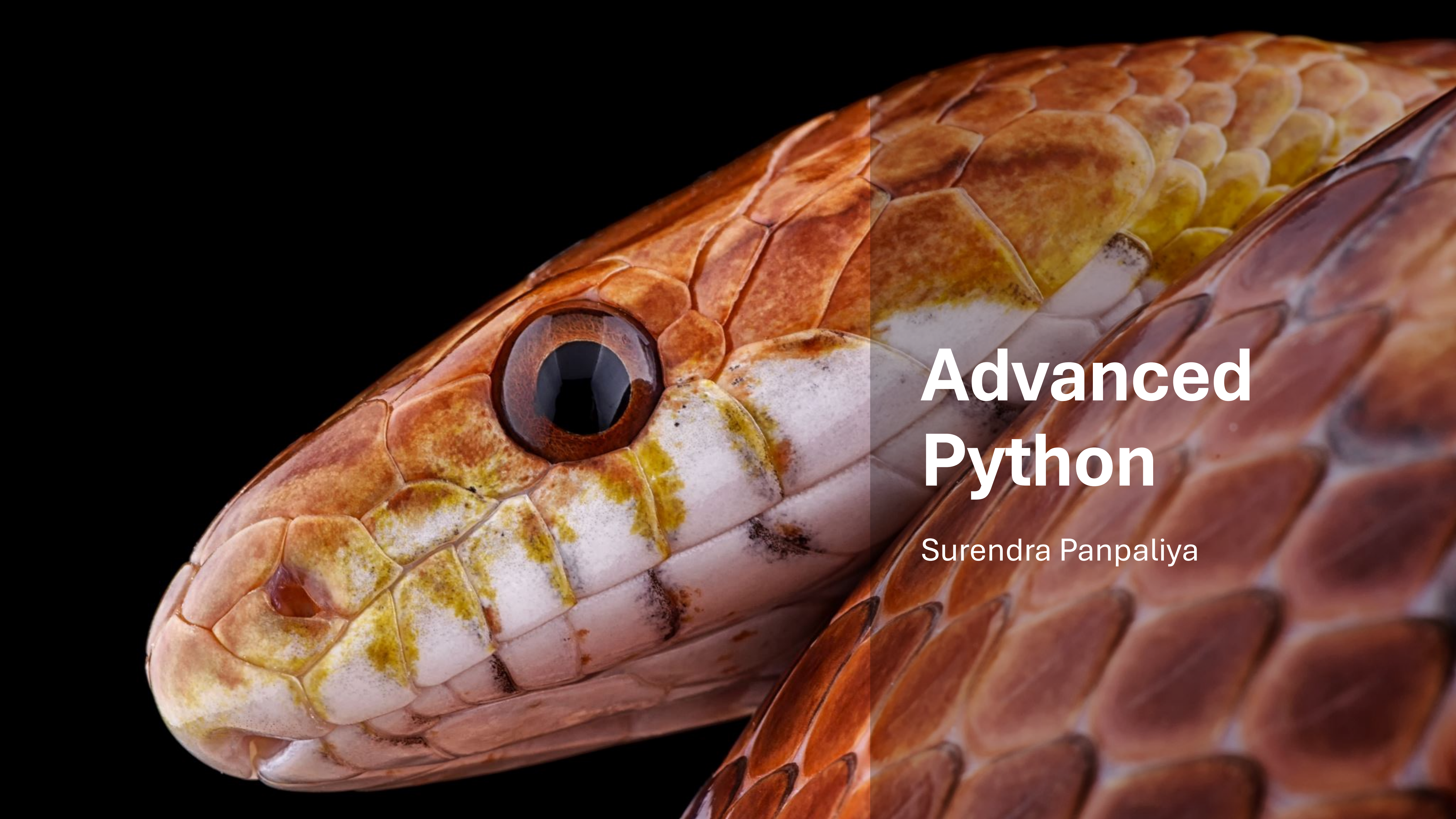


Advanced Python

Surendra Panpaliya

Week1 (Mon, Tue, Thurs)

Day 1: Python Recap + Environment & Tooling

Day 2: Functional Programming & Object-Oriented Design

Day 3: Advanced Python Concepts

Week2 (Mon, Tue, Wed, Thurs)

- Day 4: Concurrency and Async Programming
- Day 5: Web Services with FastAPI
- Day 6: Azure Functions & Cloud Deployment
- Day 7: Testing, Linting & Final Project

Day 1: Python Recap + Environment & Tooling

- Quick Python vs C# syntax mapping
- Variables, data types, control flow in Python
- Comprehensions (List, Dict, Set)
- Functions: *args, **kwargs, lambda
- Tooling: pip, venv, poetry, dependency locking

Day 1: Python Recap + Environment & Tooling

- Hands-On Lab:
- Set up Python project & create utility functions
- *Python vs C#: Static typing vs dynamic,*
- *Main method vs scripts,*
- *No semicolons/curly braces*

Day 2: Functional Programming & Object-Oriented Design

- Pythonic FP: map, filter, reduce, functools
- Object-Oriented Programming:
- C# vs Python class structure
- SOLID principles and Mixins
- Design Patterns: Singleton, Factory, Strategy

Day 2: Functional Programming & Object-Oriented Design

Hands-On Lab:

Create modular design for plugin architecture

Authentication using OAuth / Azure AD

Python vs C#: Interfaces vs duck typing,

Access modifiers

Pythonic FP: map, filter, reduce, functools

Surendra Panpaliya

🧠 What is Functional Programming?

Treats functions as **first-class citizens**

Pass them as arguments

Return them from other functions

Store them in data structures

1. map() – Apply Function to All Items

Applies a **function** to **every element** in an iterable.

map(function, iterable)

2. filter() – Filter Items Based on Condition

Returns only the elements

where the **function returns True.**

filter(function, iterable)

3. reduce() – Reduce to a Single Value

Applies a **rolling computation**

to reduce a list to a single result.

from functools import reduce

3. reduce() – Reduce to a Single Value

```
from functools import reduce

nums = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, nums)
print(product)  # Output: 24
```

functools.partial()

Fix Some Function Arguments

```
from functools import partial

def power(base, exponent):
    return base ** exponent
```

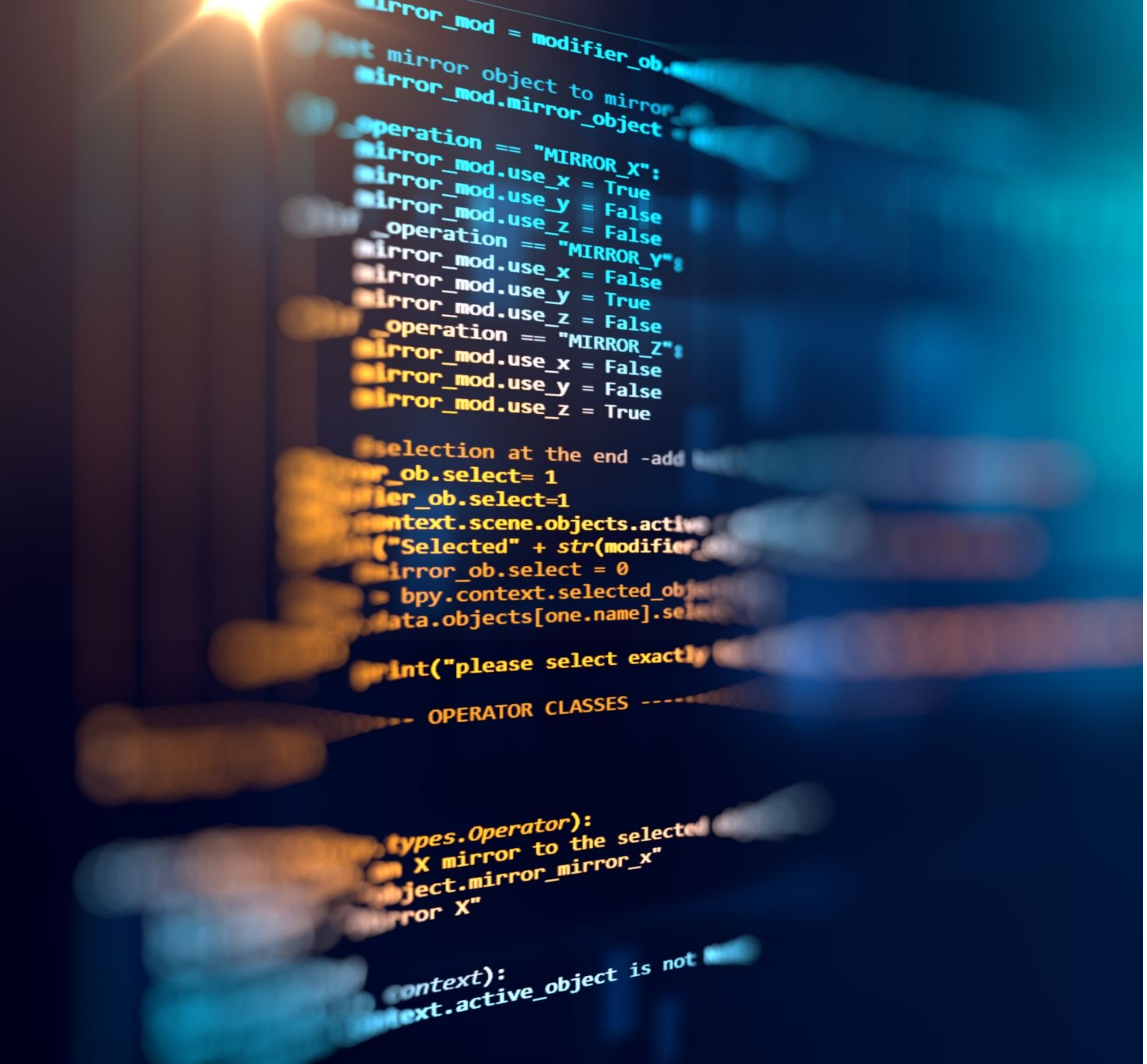
functools.partial()

```
square = partial(power, exponent=2)
cube = partial(power, exponent=3)

print(square(5))  # Output: 25
print(cube(2))    # Output: 8
```

Summary Table

Function	Purpose	Example
map()	Transform each item	map(lambda x: x*2, [1, 2, 3]) → [2, 4, 6]
filter()	Filter items	filter(lambda x: x>2, [1, 2, 3]) → [3]
reduce()	Combine into one	reduce(lambda x,y: x+y, [1,2,3]) → 6
partial()	Fix some args	partial(power, exponent=2) creates square()



Object-Oriented Programming

Surendra Panpaliya

Python vs C# class structure

Feature	Python	C#
Type System	Dynamic Typing	Static Typing
Class Definition	class ClassName:	class ClassName {}
Constructor Method	__init__()	Same name as class / constructor
Access Modifiers	Not enforced (convention-based)	Enforced (public, private, etc.)
Method Declaration	def method(self):	public void Method()
Inheritance	class B(A):	class B : A
Properties / Fields	Instance attributes via self	Defined with type in class

Python vs C# class structure

Concept	Python	C#
Constructor	__init__(self)	ClassName()
Access Modifier	_ or __ prefix (not enforced)	public, private, protected
Inheritance	class B(A):	class B : A
Static Method	@staticmethod decorator	static keyword
Method Overriding	Just redefine	Use virtual and override

Python vs C# class structure

Feature	Python	C#
Easy & Flexible	✅ Yes	❌ More structured
Strict Typing	❌ No	✅ Yes
Modern OOP	✅ Supports multiple features	✅ Rich OOP capabilities
Verbosity	🟢 Less boilerplate	🔵 More boilerplate

SOLID principles and Mixins

SOLID is an acronym for 5 principles

Make software

Easy to maintain,

Scale, and refactor.

SOLID principles and Mixins

- **S – Single Responsibility Principle (SRP)**
- **O – Open/Closed Principle (OCP)**
- **L – Liskov Substitution Principle (LSP)**
- **I – Interface Segregation Principle (ISP)**
- **D – Dependency Inversion Principle (DIP)**

S – Single Responsibility Principle (SRP)

A class should have only **one reason to change**

(i.e., one responsibility).

Each class should do just one thing and do it well.

❌ *Violation of SRP (All in one class):*

```
class Invoice:
    def calculate_total(self):
        # Calculate total logic
        pass

    def print_invoice(self):
        # Printing logic
        pass
```

❌*Violation of SRP (All in one class):*

- Invoice class has **two responsibilities**:
- Business logic (calculating total)
- Presentation logic (printing)
- If we change the printing logic
- E.g. switch from console to PDF
- Affect the business logic class

✅Following SRP (Split responsibilities)

```python
class Invoice:
    def calculate_total(self):
        # Only calculation logic
        pass

class InvoicePrinter:
    def print_invoice(self, invoice):
        # Only printing logic
        pass
```

✅Following SRP (Split responsibilities)

Invoice is responsible **only for calculation**

InvoicePrinter is responsible **only for printing**

Each class has **only one reason to change,**

which is exactly what SRP means.

Why is SRP important?

Clarity: The class has a clear purpose.

Maintainability: Changes in one responsibility won't affect unrelated parts.

Testability: Smaller classes are easier to test.

Reusability: Focused classes can be reused in different contexts.

2. O – Open/Closed Principle (OCP)

Software entities

(classes, modules, functions) should be:

Open for extension

Closed for modification

⚠️ Violation Example (Not following OCP)

```
class Discount:
    def apply(self, price, discount_type):
        if discount_type == "percentage":
            return price * 0.9
        elif discount_type == "fixed":
            return price - 50
```

⚠️ Violation Example (Not following OCP)

Every time you add a new discount type,

you have to **modify** this class.

This breaks the **Closed for modification** rule.

✅ Following OCP (Using Polymorphism)

```
class Discount:
    def apply(self, price):
        return price

class PercentageDiscount(Discount):
    def apply(self, price):
        return price * 0.9  # 10% off

class FixedDiscount(Discount):
    def apply(self, price):
        return price - 50  # ₹50 off
```

✅ Following OCP (Using Polymorphism)

- Can **add new types of discounts**
- by creating new subclasses
- (e.g., SeasonalDiscount, LoyaltyDiscount)
- **without touching the existing classes**.

✅ Following OCP (Using Polymorphism)

- The base class Discount is
- **closed for modification**
- but **open for extension**.

Real-Life Analogy

- Imagine a **smart plug**
- that supports different devices.
- You can **plug in a new device (extension)**,
- but you don’t **open and modify the plug** itself.
- That’s OCP in hardware!

Why is OCP Important?

Avoid breaking existing code

when adding new functionality.

Supports scalability

easily extend your system

without rewriting it.

Why is OCP Important?

Follows DRY

no repeated logic

when creating variations.

Facilitates unit testing

each extension can be tested in isolation.

💡 Summary

Principle	Means
Open	Allow new behavior through extension
Closed	Do not modify existing code

L – Liskov Substitution Principle (LSP)

Subtypes must be substitutable

for their base types

without altering

the correctness of the program.

L – Liskov Substitution Principle (LSP)

If class B is a subclass of class A,

you should be able to **replace A with B**

anywhere in the code

without errors or unexpected behavior.

🐦❌ Violation Example (Bird → Ostrich Problem)

```
class Bird:
    def fly(self):
        pass

class Sparrow(Bird):
    def fly(self):
        print("Flying")

class Ostrich(Bird):
    def fly(self):
        raise Exception("Ostrich can’t fly!")  # ❌ Violates LSP
```

🐦❌ Violation Example (Bird → Ostrich Problem)

- Problem:
- Ostrich is a Bird, but it **cannot** fly.
- Calling fly() on an Ostrich
- throws an error — which **breaks LSP**.

✅ Solution: Separate Behavior

```
class Bird:
    pass


class FlyingBird(Bird):
    def fly(self):
        pass
```

✅ Solution: Separate Behavior

```
class Sparrow(FlyingBird):
    def fly(self):
        print("Flying")

class Ostrich(Bird):
    def walk(self):
        print("Walking")
```

✅ Solution: Separate Behavior

- Model **flying capability only where it’s valid**.
- Sparrow is a FlyingBird, but Ostrich is **just** a Bird.
- Can substitute FlyingBird with Sparrow without issues.

❗ Why Is LSP Important?

- Ensures **reliable polymorphism**.
- Avoids **unexpected exceptions** at runtime.
- Promotes **correct inheritance** and clean design.
- Makes your code **predictable and testable**.

Summary Table

Concept	What It Means
LSP	Subclasses should work in place of their superclass
Violation	Happens when subclass behavior contradicts parent class
Solution	Use interface segregation, composition, or restructure hierarchy

I – Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods they do not use.

I – Interface Segregation Principle (ISP)

C# Example:

```
interface IPrinter { void Print(); }
interface IScanner { void Scan(); }

class MultiFunctionPrinter : IPrinter, IScanner { }
class SimplePrinter : IPrinter { } // Doesn't need Scan()
```

I – Interface Segregation Principle (ISP)

In Python,
it’s handled using
duck typing or
ABC modules.
Don’t need explicit interfaces
like in Java or C#.

I – Interface Segregation Principle (ISP)

Python relies on behavior:

If an object has a method called print()

it's treated as printable.

Don't check the type,

you check the behavior.

🚫 Problem it Solves

Unnecessary implementation

Code bloat

Violation of SRP

(class starts doing more than needed)

I – Interface Segregation Principle (ISP)

Don't need explicit interfaces

like in Java or C#.

Python relies on **behavior**

I – Interface Segregation Principle (ISP)

Object has a method print()

Treated as printable.

Don't check the type

Check the behavior

🔍❌ Violation Example (Bad Design):

```python
class Machine:
    def print(self): pass
    def scan(self): pass
    def fax(self): pass

class OldPrinter(Machine):
    def print(self): print("Printing")
    def scan(self): raise NotImplementedError("Not supported")
    def fax(self): raise NotImplementedError("Not supported")
```

🔍❌ Violation Example (Bad Design):

🛑 Problem:

OldPrinter is forced to implement

scan() and fax()

even though

it doesn’t support them.

✅ ISP-compliant Design (Using Duck Typing)

```
class IPrinter:
    def print(self):
        raise NotImplementedError

class IScanner:
    def scan(self):
        raise NotImplementedError
```

✅ ISP-compliant Design (Using Duck Typing)

```
# Only print capability
class SimplePrinter(IPrinter):
    def print(self):
        print("Simple printing")

# Full-featured
class MultiFunctionPrinter(IPrinter, IScanner):
    def print(self):
        print("Printing")

    def scan(self):
        print("Scanning")
```

✅ ISP-compliant Design (Using Duck Typing)

SimplePrinter depends only on the print() method

MultiFunctionPrinter depends on both print() and scan()

Classes aren’t burdened with unused methods ✅

✅ Alternative (Using Python’s abc module for stronger contracts)

```
from abc import ABC, abstractmethod

class IPrinter(ABC):
    @abstractmethod
    def print(self): pass

class IScanner(ABC):
    @abstractmethod
    def scan(self): pass
```

✅ Alternative (Using Python’s abc module for stronger contracts)

```
class SimplePrinter(IPrinter):
    def print(self):
        print("Simple printing")

class SmartPrinter(IPrinter, IScanner):
    def print(self):
        print("Smart printing")

    def scan(self):
        print("Smart scanning")
```

🧠 Summary

Principle	Meaning
ISP	A class should only implement what it uses
Python	Uses duck typing or ABC module to achieve this
Result	Smaller, cleaner, modular classes

Why Interface Segregation Principle Matters

Promotes **clean design.**

Encourages **role-specific interfaces.**

Prevents **unnecessary dependencies.**

D – Dependency Inversion Principle (DIP)

High-level modules should not depend on

low-level modules.

Both should depend on abstractions.

Abstractions should not depend on details.

Details should depend on abstractions.

D – Dependency Inversion Principle (DIP)

Instead of directly depending

on concrete classes (low-level),

Code should depend on

interfaces or abstract classes.

D – Dependency Inversion Principle (DIP)

Can swap implementations

without changing business logic.

System becomes

flexible, testable, and maintainable.

❌ Violation of DIP (Tight Coupling)

```
class MySQLDatabase:
    def save(self, data):
        print("Saving to MySQL")

class App:
    def __init__(self):
        self.db = MySQLDatabase()  # tightly coupled

    def save_data(self, data):
        self.db.save(data)
```

❌ Violation of DIP (Tight Coupling)

If you want to use PostgreSQL,

Need to modify the App class.

That’s bad design

breaks the Closed for Modification principle too.

✅ DIP-Compliant Example (Using Abstraction)

```
# Abstraction
class Database:
    def save(self, data):
        raise NotImplementedError

# Low-level implementation
class MySQLDatabase(Database):
    def save(self, data):
        print("Saving to MySQL")
```

✅ DIP-Compliant Example (Using Abstraction)

```
class PostgreSQLDatabase(Database):
    def save(self, data):
        print("Saving to PostgreSQL")

# High-level module
class App:
    def __init__(self, db: Database):  # depends on abstraction
        self.db = db

    def save_data(self, data):
        self.db.save(data)
```

✅ DIP-Compliant Example (Using Abstraction)

```
# Dependency Injection
app = App(MySQLDatabase())
app.save_data("data")

app2 = App(PostgreSQLDatabase())
app2.save_data("data")
```

Why Use DIP?

Benefit	Description
Flexibility	Easily switch or upgrade implementations
Testability	Can inject mock/fake classes for unit testing
Maintainability	Changes in low-level modules don't affect high-level modules
Scalability	Extend functionality with minimal changes

🎯 Real-World Analogy

Imagine an electric socket (abstraction).

You can plug in a phone charger,

laptop adapter, or lamp

without changing the wall socket.

The socket **depends on an interface**

not the actual device.

🛠️ Example with Dependency Injection for Unit Testing

```
class MockDatabase(Database):
    def save(self, data):
        print(f"[Mock] Pretending to save: {data}")


# In test
test_app = App(MockDatabase())
test_app.save_data("test-data")
```

Summary Table

Concept	Explanation
High-Level Module	Business logic (App)
Low-Level Module	Database implementations
Abstraction	Interface (Database)
DIP	High-level depends on interface, not concrete classes

Mixins (Python Only)

A class that **adds extra**

functionality to another class

via **multiple inheritance,**

but **doesn’t stand alone.**

When to Use?

You want to add reusable methods

to multiple classes.

You don't want to create

deep class hierarchies.

Key Mixin Rules

Do not define __init__ in mixins.

Keep mixins **narrow and focused.**

Combine with other base classes

via multiple inheritance.

Summary

Concept	Python Support	C# Support	Notes
SRP	✅	✅	Keep classes focused
OCP	✅	✅	Use inheritance or composition
LSP	✅	✅	Use proper base classes
ISP	✅ (via duck typing)	✅	Use small interfaces
DIP	✅	✅	Use dependency injection
Mixins	✅	❌	Only in dynamic languages like Python

Design Patterns: Singleton, Factory, Strategy

Surendra Panpaliya

What is a Design Pattern?

Reusable solutions

to **common problems**

that occur in software design.

What is a Design Pattern?

LIKE **PROVEN TEMPLATES OR BLUEPRINTS**

USED BY DEVELOPERS TO SOLVE PROBLEMS

EFFICIENTLY AND CONSISTENTLY

📚 Why Use Design Patterns?

Benefit	Explanation
🔁 Reusability	Saves time by reusing proven solutions
📐 Consistency	Brings standard structure to the codebase
🧠 Readability	Makes the design more understandable to other developers
🧪 Testability	Encourages decoupled, testable code
🔄 Flexibility	Easier to modify, extend, or replace components

Categories of Design Patterns

Creational – Object creation

(e.g., Singleton, Factory)

Structural – Class and object composition

(e.g., Adapter, Decorator)

Categories of Design Patterns

Behavioral

Object interaction and responsibility

(e.g., Strategy, Observer)

Singleton Pattern – Creational Pattern

Ensures a class has

only one instance

provides a **global access point**

to that instance.

Singleton Pattern – Creational Pattern

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance
```

Singleton Pattern – Creational Pattern

```
# Usage
obj1 = Singleton()
obj2 = Singleton()

print(obj1 is obj2)  # True – Same instance
```

Use Cases

- Configuration managers
- Logging
- Database connections
- Cache

Factory Pattern– Creational Pattern

Creates objects without exposing

the instantiation logic to the client.

Let subclasses or methods decide

which class to instantiate.

Factory Pattern– Creational Pattern

```
class Shape:
   def draw(self): pass

class Circle(Shape):
   def draw(self):
      print("Drawing Circle")

class Square(Shape):
   def draw(self):
      print("Drawing Square")
```

Factory Pattern– Creational Pattern

```
class ShapeFactory:
    def get_shape(self, shape_type):
        if shape_type == "circle":
            return Circle()
        elif shape_type == "square":
            return Square()

# Usage
factory = ShapeFactory()
shape = factory.get_shape("circle")
shape.draw()  # Drawing Circle
```

📦 Use Cases

GUI FRAMEWORKS

PARSER OBJECTS

GAME OBJECT CREATION

Strategy Pattern – Behavioral Pattern

Defines a family of algorithms,

encapsulates each one, and

makes them interchangeable.

The strategy lets the algorithm vary

independently from clients.

Strategy Pattern – Behavioral Pattern

```
class PaymentStrategy:
    def pay(self, amount): pass

class CreditCardPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paid ₹{amount} using Credit Card.")

class PayPalPayment(PaymentStrategy):
    def pay(self, amount):
        print(f"Paid ₹{amount} using PayPal.")
```

Strategy Pattern – Behavioral Pattern

```
class ShoppingCart:
    def __init__(self, strategy: PaymentStrategy):
        self.strategy = strategy

    def checkout(self, amount):
        self.strategy.pay(amount)
```

Strategy Pattern – Behavioral Pattern

```
# Usage
cart = ShoppingCart(CreditCardPayment())
cart.checkout(500)

cart.strategy = PayPalPayment()
cart.checkout(300)
```

📦 Use Cases

- Payment methods
- Sorting algorithms
- Compression strategies (e.g., ZIP, RAR)
- Navigation (e.g., driving, walking)

Summary Table

Pattern	Type	Purpose
Singleton	Creational	Ensure one instance
Factory	Creational	Delegate object creation
Strategy	Behavioral	Switchable algorithms at runtime

Summary Table

Pattern	Purpose	Real-World Use
Singleton	Ensure only one instance exists	Config Manager, Logger
Factory	Create objects without specifying the exact class	Notification Service, Shape Creator
Strategy	Swap behavior (algorithms) at runtime	Payment Gateway, Sorting Logic

Happy Learning !! Thanks for Your Patience ☺

Surendra Panpaliya

GKTCS Innovations