Check out service:

Jab user ya customer koi cheez select karne ke baad **payment aur confirmation ka process complete karta hai**, usay **check out** kehte hain.

🛒 Example (Online Shopping):

Jab aap cart mein products dal leti hain, aur **"Proceed to Checkout"** pe click karti hai wahan se address, payment method, aur final order place hota hai — **yeh pura process check out service hota hai**.

Work flow:

Ek step-by-step process jisme koi kaam complete hota hai — har step ke baad agla step fix hota hai.

📚 Simple Example (Real Life):

Tea Banana (Chai ka Workflow):

1. Pani garam karo
2. Chai patti dalo
3. Doodh dalo
4. Ubalne do
5. Cup mein dalo

☞ Ye **chai banane ka workflow** hai.

Orchestrator: اورکسٹریشن

"Orchestrator woh hota hai jo multiple tasks ya workflows ko coordinate aur automate karta hai taake sab steps sahi sequence mein, sahi timing se aur efficiently complete ho jaayein."

MRO (Method Resolution Order):

"Agar ek child class bohot saari classes se inherit kar rahi hai, toh Python kis order mein un parent classes ka method dhoondhega aur use karega — uss order ko MRO kehte hain."

📃 Jab zarurat hoti hai?

Jab aap **Multiple Inheritance** use karte ho (ek se zyada parent class inherit karni ho), tab MRO kaafi important hoti hai.

🔧 CODE Example:

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

class B(A):
    def show(self):
        print("B class")

class C(A):
    def show(self):
        print("C class")

class D(B, C):  # Multiple inheritance
    pass

obj = D()
obj.show()

print(D.mro())  # Method Resolution Order
```

```
B class
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class
'object'>]
```

MRO ka Rule Left to Right:

Jo class pehle likhi hoti hai inheritance mein, uska method pehle check hota hai.

`class D(B, C)`

→ Pehle `B` ka method dekha, phir `C`, phir `A`.

`__init__()`:

`__init__()` **constructor method** hota hai,
Aur **ye Python ke built-in class `object` ka method hota hai**,
Jo **automatically chalta hai jab bhi aap kisi class ka object banate ho**.

`__init__()` static method nahi hota.
Ye ek **instance method** hota hai jo **self parameter** leta hai.

- `__new__()` ye ek **static method** hota hai jo object banata hai.

```
class MyClass:
    def __new__(cls, *args, **kwargs):
        print("1. __new__ called")  # Step 1
        instance = super().__new__(cls)  # object banana
        return instance

    def __init__(self):
        print("2. __init__ called")  # Step 2

obj = MyClass()
```

```
1. __new__ called
2. __init__ called
```

Singleton Pattern:

Sirf **ek hi object** banane dena hai — to `__new__()` se check karte hain pehle se bana hai ya nahi.

```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            print("Creating new instance")
            cls._instance = super().__new__(cls)
        return cls._instance

obj1 = Singleton()
obj2 = Singleton()

print(obj1 is obj2)  # True (same object)
```

Return:

`return` ek **Python keyword** hai

☞ Function se **result** ya **value wapas (return)** karna

```
def add(a, b):
    return a + b

result = add(5, 3)
print(result)

8
```

Ye function se value ko wapas karta hai taake hum usay baad mein use kar saken. Agar return na ho to function None return karta hai.

Jab hum function ko call karte hain, kya hota hai?

Socho Python ek memory system (stack) rakhta hai — jise hum Call Stack kehte hain.

Step 1: Function ka ek naya box (frame) banata hai memory mein

Step 2: Us frame ke andar sab variables store karta hai

Step 3: Jab return hota hai → wo frame delete ho jata hai (close ho jata hai)

Function ko jab call karo	Python ek temporary memory area banata hai (isko Stack Frame kehte hain)
Function ke andar variables	Isi frame mein store hote hain
Jab return hota hai	Frame khatam ho jata hai, aur result wapas chala jata hai jahan se function call hua tha

```
def add(a, b):
   result = a + b
   return result

x = add(2, 3)
print(x)
```

1. S - Single Responsibility Principle (SRP):

Ek class sirf ek hi kaam kare. Har class ka sirf ek *zimmedari* (responsibility) ho.

Yani agar code mein koi badlaav aata hai, toh class sirf ek hi wajah se change honi chahiye.

🎁 Real Life Example (Urdu Main):

Ek **Printer** machine lo:

- Agar ek hi class print kare, scan kare, fax bheje, copy kare — toh woh class bohat mushkil ho jati hai maintain karna.
- Agar har kaam alag class ko diya jaye (PrintHandler, ScanHandler, FaxHandler) — toh agar fax ka code change karna ho, toh sirf **FaxHandler** class badlegi, baaki nahi.

✅ Python Example: Bad SRP (Violation)

```
class Report:
   def __init__(self, content):
      self.content = content

   def save_to_file(self, filename):
      with open(filename, 'w') as file:
         file.write(self.content)

   def print_report(self):
      print(self.content)
```

☠️ Kya masla hai?

- Yeh class **data store** bhi kar rahi hai

- Aur **file save** aur **printing** bhi
- 3 alag zimmedariyan! → SRP ka violation

✅ **Python Example: Good SRP (Followed)**

```
class Report:
    def __init__(self, content):
        self.content = content

class ReportPrinter:
    def print(self, report):
        print(report.content)

class ReportSaver:
    def save_to_file(self, report, filename):
        with open(filename, 'w') as file:
            file.write(report.content)
```

- Ab agar save method mein bug ho, sirf `ReportSaver` class badlegi.
- `Report` class ko chhoone ki zarurat nahi.

2. O - Open/Closed Principle (OCP)

Code ko aap extend (bara) kar sako, lekin usay modify (badalna) na pade."

- Jab new feature add karo, purane code ko **nahi** chhedna chahiye.
- Instead, aap **naya code add** karo (inheritance ya polymorphism se).

```
from abc import ABC, abstractmethod

class PaymentMethod(ABC):
    @abstractmethod
    def pay(self):
        pass
```

```
class CreditCard(PaymentMethod):
    def pay(self):
        print("Processing credit card")

class PayPal(PaymentMethod):
    def pay(self):
        print("Processing PayPal")

# Ab hum new method asani se add kar sakty hain
class JazzCash(PaymentMethod):
    def pay(self):
        print("Processing JazzCash")

# Usage:
def make_payment(method: PaymentMethod):
    method.pay()

make_payment(CreditCard())
make_payment(PayPal())
make_payment(JazzCash())  # new method, no old code change
```

Code ko badlay bina naya feature add kar sako.

3. Liskov Substitution Principle (LSP) – One Line:

Jo kaam parent class kare, wahi kaam child class bhi sahi tareeke se kare."

```
class Bird:
    def fly(self):
        print("Flying in the sky!")

class Sparrow(Bird):
```

```
    pass

class Ostrich(Bird):
    def fly(self):
        raise Exception("Ostrich cannot fly!")  # ⚠ violates LSP
```

Problem: `Ostrich` ko `Bird` ke jesa treat kiya, lekin woh fly nahi kar sakta — is se program crash ho sakta hai. ✗

✓ Viva Answer:

"Liskov Principle kehta hai ke child class, parent class ki jagah use ho sakay bina code ko toray."

4. Interface Segregation Principle (ISP) – One Line:

"Client ko sirf wohi methods dena chahiye jo use chahiyein, extra nahi."

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

class OldPrinter(Machine):
    def print(self): pass
    def scan(self): raise NotImplementedError()
    def fax(self): raise NotImplementedError()
```

Problem: `OldPrinter` sirf print karta hai — lekin usse scan aur fax ke methods bhi mil gaye jo usko chahiye hi nahi. ✗

✓ Solution (Right ✓):

```
class Printer:
   def print(self): pass

class Scanner:
   def scan(self): pass

class Fax:
   def fax(self): pass

class OldPrinter(Printer):  # sirf print use kare
   def print(self): pass
```

ISP kehta hai ke class ko sirf wohi interface (methods) milne chahiyein jo uske kaam ke hoon

Dependency Inversion Principle (DIP) – One Line:

Bari classes aur choti classes direct ek dosray pr depend na hon — dono ko abstract (interface) ke zariye connect karo."

📌 Problem Example (Wrong ❌):

```
class LightBulb:
   def turn_on(self):
      print("Light ON")

   def turn_off(self):
      print("Light OFF")

class Switch:
   def __init__(self):
      self.bulb = LightBulb()  # direct dependency ❌

   def operate(self):
      self.bulb.turn_on()
```

Issue: `Switch` class directly `LightBulb` pr depend kr rahi hai. Agar bulb change ho gaya (fan, heater), to Switch class bhi change karni padegi ❌

✅ Better Solution Using Abstraction (Right ✅):

```
class SwitchableDevice:  # abstraction
    def turn_on(self): pass
    def turn_off(self): pass

class LightBulb(SwitchableDevice):
    def turn_on(self):
        print("Light ON")
    def turn_off(self):
        print("Light OFF")

class Switch:
    def __init__(self, device: SwitchableDevice):  # abstraction
        self.device = device

    def operate(self):
        self.device.turn_on()
```

Ab `Switch` ko farq nahi padta ke wo Light chala raha hai ya Fan — abstraction (interface) ke zariye kaam ho raha hai ✅

✅ Viva Answer:

"DIP kehta hai ke high-level aur low-level classes directly ek dosray pr depend na karein. Unke darmiyan abstraction ka layer hona chahiye."

****