#### Template Method模式示例：

from abc import ABC, abstractmethod

class DataProcessor(ABC):

# 模板方法

def process\_data(self):

self.load\_data()

self.transform\_data()

if self.need\_validation():

self.validate\_data()

self.save\_results()

# 具体方法（已有默认实现）

def load\_data(self):

print("Loading data from default source...")

# 抽象方法（必须由子类实现）

@abstractmethod

def transform\_data(self):

pass

# 钩子方法（可选覆盖）

def need\_validation(self):

return False

def validate\_data(self):

print("Performing default validation...")

@abstractmethod

def save\_results(self):

pass

class CSVProcessor(DataProcessor):

def transform\_data(self):

print("Transforming CSV data: parsing, cleaning, etc.")

def need\_validation(self):

return True # 覆盖钩子方法

def validate\_data(self):

print("Performing CSV-specific validation checks")

def save\_results(self):

print("Saving results to CSV file")

class JSONProcessor(DataProcessor):

def transform\_data(self):

print("Transforming JSON data: decoding, schema mapping")

def save\_results(self):

print("Uploading JSON results to API endpoint")

# 使用示例

if \_\_name\_\_ == "\_\_main\_\_":

print("--- Processing CSV ---")

csv\_processor = CSVProcessor()

csv\_processor.process\_data()

print("\n--- Processing JSON ---")

json\_processor = JSONProcessor()

json\_processor.process\_data()

#### 特点总结：

结构固定行为可变：定义算法骨架，子类重写特定步骤。

代码复用：公共逻辑在基类中实现。

扩展控制：

抽象方法强制子类实现。

钩子方法允许可选扩展。

好莱坞原则："Don't call us, we'll call you" - 子类不直接调用基类。

应用场景：数据处理管道、游戏主循环、UI框架生命周期等。

#### Singleton模式示例：

class DatabaseConnection:

\_instance = None

def \_\_new\_\_(cls):

if cls.\_instance is None:

print("Creating new database connection")

cls.\_instance = super().\_\_new\_\_(cls)

cls.\_instance.initialize()

return cls.\_instance

def initialize(self):

self.connection\_count = 0

self.status = "DISCONNECTED"

def connect(self):

if self.status == "DISCONNECTED":

self.connection\_count += 1

self.status = "CONNECTED"

print(f"Established connection #{self.connection\_count}")

return self

def execute(self, query):

print(f"Executing: {query}")

def \_\_str\_\_(self):

return f"<DB Connection {id(self)} | Status: {self.status}>"

# 测试单例行为

if \_\_name\_\_ == "\_\_main\_\_":

# 多次获取实例

db1 = DatabaseConnection().connect()

db2 = DatabaseConnection().connect()

print(f"\nID check: {id(db1) == id(db2)}") # True

print(f"Status: {db1.status}") # CONNECTED

# 测试状态共享

db1.execute("SELECT \* FROM users")

db2.status = "MAINTENANCE"

print(f"db1 status: {db1.status}") # MAINTENANCE

# 尝试直接实例化

try:

db3 = DatabaseConnection.\_\_new\_\_(DatabaseConnection)

except Exception as e:

print(f"\nPrevented direct creation: {type(e).\_\_name\_\_}")

#### 特点总结：

全局唯一实例：

new控制实例创建

类变量\_instance存储唯一实例

资源保护：

防止多次连接开销

避免状态冲突，如配置文件修改冲突

线程安全（需根据语言扩展）：

Python示例在单线程下安全

多线程需加锁

访问控制：

全局访问点，直接调用类

防止反射/序列化破坏，如示例中new防护

应用场景：数据库连接池、配置管理器、设备驱动访问、日志记录器

#### Strategy Pattern模式示例：

from abc import ABC, abstractmethod

# 支付策略接口

class PaymentStrategy(ABC):

@abstractmethod

def pay(self, amount: float) -> str:

pass

# 具体策略类：信用卡支付

class CreditCardPayment(PaymentStrategy):

def \_\_init\_\_(self, card\_number, cvv):

self.card\_number = card\_number

self.cvv = cvv

def pay(self, amount: float) -> str:

return f"Paid ${amount} using Credit Card ending in {self.card\_number[-4:]}"

# 具体策略类：PayPal支付

class PayPalPayment(PaymentStrategy):

def \_\_init\_\_(self, email):

self.email = email

def pay(self, amount: float) -> str:

return f"Paid ${amount} using PayPal account {self.email}"

# 具体策略类：加密货币支付

class CryptoPayment(PaymentStrategy):

def \_\_init\_\_(self, wallet\_address):

self.wallet\_address = wallet\_address

def pay(self, amount: float) -> str:

return f"Paid ${amount} using Crypto wallet {self.wallet\_address[:8]}..."

# 上下文类：购物车

class ShoppingCart:

def \_\_init\_\_(self, payment\_strategy: PaymentStrategy):

self.\_payment\_strategy = payment\_strategy

self.\_items = []

def add\_item(self, item, price):

self.\_items.append((item, price))

def total\_price(self) -> float:

return sum(price for \_, price in self.\_items)

def checkout(self) -> str:

total = self.total\_price()

return self.\_payment\_strategy.pay(total)

def set\_payment\_strategy(self, strategy: PaymentStrategy):

self.\_payment\_strategy = strategy

# 使用示例

if \_\_name\_\_ == "\_\_main\_\_":

# 创建支付策略

credit\_card = CreditCardPayment("1234567812345678", "123")

paypal = PayPalPayment("user@example.com")

# 创建购物车

cart = ShoppingCart(credit\_card)

cart.add\_item("Laptop", 1200)

cart.add\_item("Mouse", 50)

# 使用默认策略支付

print("Checkout 1:", cart.checkout())

# 动态切换支付策略

cart.set\_payment\_strategy(paypal)

print("Checkout 2:", cart.checkout())

# 添加新策略无需修改购物车代码

crypto = CryptoPayment("0x1a2b3c4d5e6f7a8b9c")

cart.set\_payment\_strategy(crypto)

cart.add\_item("USB Drive", 20)

print("Checkout 3:", cart.checkout())

#### 特点总结：

灵活替换算法：运行时动态切换策略对象。

消除条件分支：避免冗长的if-else或switch-case语句。

开闭原则支持：可扩展新策略而无需修改上下文代码。

算法封装：将算法细节与其使用环境分离。

增加对象数量：每个策略需要单独的类实现。

#### Factory Method Pattern模式示例：

from abc import ABC, abstractmethod

# 产品接口：交通工具

class Vehicle(ABC):

@abstractmethod

def deliver(self, destination: str) -> str:

pass

# 具体产品：卡车

class Truck(Vehicle):

def deliver(self, destination: str) -> str:

return f"卡车陆运交付到 {destination}"

# 具体产品：轮船

class Ship(Vehicle):

def deliver(self, destination: str) -> str:

return f"轮船海运交付到 {destination}"

# 具体产品：飞机

class Airplane(Vehicle):

def deliver(self, destination: str) -> str:

return f"飞机空运交付到 {destination}"

# 创建者接口：物流中心

class LogisticsCenter(ABC):

# 工厂方法 (核心)

@abstractmethod

def create\_vehicle(self) -> Vehicle:

pass

def plan\_delivery(self, destination: str) -> str:

vehicle = self.create\_vehicle()

result = vehicle.deliver(destination)

return f"{self.\_\_class\_\_.\_\_name\_\_}: {result}"

# 具体创建者：陆地物流中心

class RoadLogistics(LogisticsCenter):

def create\_vehicle(self) -> Vehicle:

print("准备卡车...检查油量...装载货物...")

return Truck()

# 具体创建者：海运物流中心

class SeaLogistics(LogisticsCenter):

def create\_vehicle(self) -> Vehicle:

print("准备轮船...检查集装箱...安排船员...")

return Ship()

# 具体创建者：空运物流中心

class AirLogistics(LogisticsCenter):

def create\_vehicle(self) -> Vehicle:

print("准备飞机...安全检测...清点货物...")

return Airplane()

# 使用示例

if \_\_name\_\_ == "\_\_main\_\_":

# 选择物流方式

def deliver\_package(logistics: LogisticsCenter, destination: str):

print(logistics.plan\_delivery(destination))

# 根据不同需求选择不同物流中心

deliver\_package(RoadLogistics(), "北京仓库")

deliver\_package(SeaLogistics(), "上海港口")

deliver\_package(AirLogistics(), "纽约机场")

# 添加新型物流方式无需修改客户端代码

class RailLogistics(LogisticsCenter):

def create\_vehicle(self) -> Vehicle:

print("准备火车...检查车厢...安排班次...")

# 新增加的铁路交通工具

class Train(Vehicle):

def deliver(self, destination: str) -> str:

return f"火车铁运交付到 {destination}"

return Train()

deliver\_package(RailLogistics(), "成都火车站")

#### 特点总结：

解耦创建与使用：客户端只依赖抽象接口，不依赖具体类。

扩展性强：添加新产品只需新增工厂子类。

单一职责：每个工厂负责创建单一产品类型。

多态性支持：客户端通过基本接口处理所有产品。

增加类数量：每个产品需要对应的工厂类。

#### Template Method

核心优点：​​

流程统一控制：提供清晰的算法结构框架，确保核心流程稳定。

代码复用性强：公共逻辑集中在父类，子类只需关注差异点实现。

扩展点明确：通过抽象方法和钩子方法定义扩展边界。

避免重复：解决相似实现中代码冗余问题。

主要局限：​​

继承依赖固化：子类被绑定到特定父类结构，增加耦合度。

扩展受限：算法流程固定，无法动态改变步骤顺序。

类膨胀风险：每个变体都需要创建新子类。

设计僵化：过度使用时导致灵活性下降。

最佳适用场景： 当多个组件有固定流程但部分实现差异时，如数据导入流程、游戏角色行为树。

#### Singleton

核心优点：​​

资源高效管理：避免重复创建资源密集型对象。

状态一致性：全局共享状态确保数据统一。

访问便捷性：提供统一访问入口。

内存优化：减少内存占用和GC压力。

主要局限：​​

测试困难：全局状态难以隔离测试。

隐藏依赖：可能导致模块之间隐性耦合。

生命周期管理复杂：过早初始化或未释放问题。

违反单一职责：兼具实例管理和业务功能。

最佳适用场景：真正需要全局唯一访问点的资源，如数据库连接池、应用程序配置。

#### Strategy

核心优点：​​

运行时灵活性：支持动态切换算法实现。

解耦彻底：将算法实现与使用环境完全分离。

开闭原则典范：新增策略不修改已有代码。

消除条件分支：替代复杂if-else/switch结构。

主要局限：​​

对象数量膨胀：每个策略需要单独类实现。

客户认知负担：需理解不同策略适用场景。

性能开销：额外的对象创建和方法调用。

过度抽象风险：简单逻辑可能设计过度。

最佳适用场景：存在多种可互换算法，如支付方式、排序算法、折扣策略。

#### Factory Method

核心优点：​​

创建解耦：分离对象创建与使用逻辑。

扩展无缝：新增产品类型不影响客户代码。

多态支持：客户端通过接口处理产品。

子类控制：将实例化决策权交给子类。

主要局限：​​

类激增问题：每个产品需对应工厂类。

设计复杂性：层级结构增加理解难度。

扩展限制：新增产品必须修改工厂类。

间接访问：多一层抽象影响性能。

最佳适用场景：对象创建逻辑复杂或需要支持多系列产品，如UI组件、跨平台适配。