**Python Design Patterns Overview**

Design patterns are reusable solutions to common problems in software design. In Python, design patterns are implemented using its object-oriented features, functional programming capabilities, and dynamic typing. Below are some commonly used design patterns with example code:  
  
---  
  
## 1. \*\*Singleton Pattern\*\*  
The Singleton pattern ensures that a class has only one instance and provides a global point of access to it.  
  
```python  
class Singleton:  
 \_instance = None  
  
 def \_\_new\_\_(cls, \*args, \*\*kwargs):  
 if not cls.\_instance:  
 cls.\_instance = super(Singleton, cls).\_\_new\_\_(cls, \*args, \*\*kwargs)  
 return cls.\_instance  
  
# Example usage  
singleton1 = Singleton()  
singleton2 = Singleton()  
  
print(singleton1 is singleton2) # True  
```  
  
---  
  
## 2. \*\*Factory Pattern\*\*  
The Factory pattern provides a way to create objects without specifying the exact class of the object that will be created.  
  
```python  
class Shape:  
 def draw(self):  
 pass  
  
class Circle(Shape):  
 def draw(self):  
 return "Drawing a Circle"  
  
class Square(Shape):  
 def draw(self):  
 return "Drawing a Square"  
  
class ShapeFactory:  
 @staticmethod  
 def get\_shape(shape\_type):  
 if shape\_type == "circle":  
 return Circle()  
 elif shape\_type == "square":  
 return Square()  
 else:  
 return None  
  
# Example usage  
shape = ShapeFactory.get\_shape("circle")  
print(shape.draw()) # Drawing a Circle  
```  
  
---  
  
## 3. \*\*Observer Pattern\*\*  
The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified.  
  
```python  
class Subject:  
 def \_\_init\_\_(self):  
 self.\_observers = []  
  
 def attach(self, observer):  
 self.\_observers.append(observer)  
  
 def detach(self, observer):  
 self.\_observers.remove(observer)  
  
 def notify(self):  
 for observer in self.\_observers:  
 observer.update()  
  
class Observer:  
 def update(self):  
 pass  
  
class ConcreteObserver(Observer):  
 def \_\_init\_\_(self, name):  
 self.name = name  
  
 def update(self):  
 print(f"{self.name} has been notified!")  
  
# Example usage  
subject = Subject()  
observer1 = ConcreteObserver("Observer 1")  
observer2 = ConcreteObserver("Observer 2")  
  
subject.attach(observer1)  
subject.attach(observer2)  
  
subject.notify()  
# Output:  
# Observer 1 has been notified!  
# Observer 2 has been notified!  
```  
  
---  
  
## 4. \*\*Decorator Pattern\*\*  
The Decorator pattern allows behavior to be added to an object dynamically.  
  
```python  
def bold\_decorator(func):  
 def wrapper(\*args, \*\*kwargs):  
 return f"<b>{func(\*args, \*\*kwargs)}</b>"  
 return wrapper  
  
def italic\_decorator(func):  
 def wrapper(\*args, \*\*kwargs):  
 return f"<i>{func(\*args, \*\*kwargs)}</i>"  
 return wrapper  
  
@bold\_decorator  
@italic\_decorator  
def greet(name):  
 return f"Hello, {name}"  
  
# Example usage  
print(greet("Alice")) # <b><i>Hello, Alice</i></b>  
```  
  
---  
  
## 5. \*\*Strategy Pattern\*\*  
The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable.  
  
```python  
class Strategy:  
 def execute(self, a, b):  
 pass  
  
class AddStrategy(Strategy):  
 def execute(self, a, b):  
 return a + b  
  
class SubtractStrategy(Strategy):  
 def execute(self, a, b):  
 return a - b  
  
class Context:  
 def \_\_init\_\_(self, strategy):  
 self.\_strategy = strategy  
  
 def set\_strategy(self, strategy):  
 self.\_strategy = strategy  
  
 def execute\_strategy(self, a, b):  
 return self.\_strategy.execute(a, b)  
  
# Example usage  
context = Context(AddStrategy())  
print(context.execute\_strategy(5, 3)) # 8  
  
context.set\_strategy(SubtractStrategy())  
print(context.execute\_strategy(5, 3)) # 2  
```  
  
---  
  
## 6. \*\*Builder Pattern\*\*  
The Builder pattern separates the construction of a complex object from its representation.  
  
```python  
class Car:  
 def \_\_init\_\_(self):  
 self.make = None  
 self.model = None  
 self.year = None  
  
 def \_\_str\_\_(self):  
 return f"{self.year} {self.make} {self.model}"  
  
class CarBuilder:  
 def \_\_init\_\_(self):  
 self.car = Car()  
  
 def set\_make(self, make):  
 self.car.make = make  
 return self  
  
 def set\_model(self, model):  
 self.car.model = model  
 return self  
  
 def set\_year(self, year):  
 self.car.year = year  
 return self  
  
 def build(self):  
 return self.car  
  
# Example usage  
builder = CarBuilder()  
car = builder.set\_make("Toyota").set\_model("Corolla").set\_year(2023).build()  
print(car) # 2023 Toyota Corolla  
```  
  
---  
  
## 7. \*\*Command Pattern\*\*  
The Command pattern encapsulates a request as an object, thereby allowing users to parameterize clients with different requests.  
  
```python  
class Command:  
 def execute(self):  
 pass  
  
class LightOnCommand(Command):  
 def execute(self):  
 print("The light is ON")  
  
class LightOffCommand(Command):  
 def execute(self):  
 print("The light is OFF")  
  
class RemoteControl:  
 def \_\_init\_\_(self):  
 self.\_commands = []  
  
 def add\_command(self, command):  
 self.\_commands.append(command)  
  
 def execute\_commands(self):  
 for command in self.\_commands:  
 command.execute()  
  
# Example usage  
remote = RemoteControl()  
remote.add\_command(LightOnCommand())  
remote.add\_command(LightOffCommand())  
  
remote.execute\_commands()  
# Output:  
# The light is ON  
# The light is OFF  
```  
  
---  
  
These are just a few examples of design patterns in Python. Each pattern is suited for specific scenarios, and understanding them can help you write more maintainable and scalable code. Let me know if you'd like to explore more patterns!

**with real-world use cases**

---  
  
### 1. \*\*Singleton Pattern\*\*  
\*\*Use Case:\*\* Managing a single instance of a resource, such as a database connection or a configuration manager.  
  
\*\*Real-World Example:\*\*  
Imagine a web application where multiple parts of the system need access to a single database connection. Instead of creating multiple connections (which is resource-intensive), the Singleton pattern ensures that only one connection instance is created and shared.  
  
```python  
class DatabaseConnection:  
 \_instance = None  
  
 def \_\_new\_\_(cls, \*args, \*\*kwargs):  
 if not cls.\_instance:  
 cls.\_instance = super(DatabaseConnection, cls).\_\_new\_\_(cls, \*args, \*\*kwargs)  
 return cls.\_instance  
  
# Example usage  
db1 = DatabaseConnection()  
db2 = DatabaseConnection()  
print(db1 is db2) # True (both are the same instance)  
```  
  
---  
  
### 2. \*\*Factory Pattern\*\*  
\*\*Use Case:\*\* Creating objects without specifying their exact class, such as in a GUI framework where different types of buttons (e.g., WindowsButton, MacButton) are created based on the operating system.  
  
\*\*Real-World Example:\*\*  
A document editor might need to create different types of documents (e.g., Word, PDF, Excel). The Factory pattern can decide which document type to create based on user input.  
  
```python  
class Document:  
 def open(self):  
 pass  
  
class WordDocument(Document):  
 def open(self):  
 return "Opening a Word document"  
  
class PDFDocument(Document):  
 def open(self):  
 return "Opening a PDF document"  
  
class DocumentFactory:  
 @staticmethod  
 def create\_document(doc\_type):  
 if doc\_type == "word":  
 return WordDocument()  
 elif doc\_type == "pdf":  
 return PDFDocument()  
  
# Example usage  
doc = DocumentFactory.create\_document("pdf")  
print(doc.open()) # Opening a PDF document  
```  
  
---  
  
### 3. \*\*Observer Pattern\*\*  
\*\*Use Case:\*\* Notifying multiple objects when the state of one object changes, such as in a stock market application where multiple clients need updates when stock prices change.  
  
\*\*Real-World Example:\*\*  
A weather station broadcasts temperature updates to multiple devices (e.g., phones, tablets, and computers). When the temperature changes, all devices are notified.  
  
```python  
class WeatherStation:  
 def \_\_init\_\_(self):  
 self.\_observers = []  
  
 def add\_observer(self, observer):  
 self.\_observers.append(observer)  
  
 def remove\_observer(self, observer):  
 self.\_observers.remove(observer)  
  
 def notify\_observers(self, temperature):  
 for observer in self.\_observers:  
 observer.update(temperature)  
  
class Device:  
 def update(self, temperature):  
 pass  
  
class Phone(Device):  
 def update(self, temperature):  
 print(f"Phone: Temperature updated to {temperature}°C")  
  
class Tablet(Device):  
 def update(self, temperature):  
 print(f"Tablet: Temperature updated to {temperature}°C")  
  
# Example usage  
station = WeatherStation()  
phone = Phone()  
tablet = Tablet()  
  
station.add\_observer(phone)  
station.add\_observer(tablet)  
  
station.notify\_observers(25)  
# Output:  
# Phone: Temperature updated to 25°C  
# Tablet: Temperature updated to 25°C  
```  
  
---  
  
### 4. \*\*Decorator Pattern\*\*  
\*\*Use Case:\*\* Dynamically adding functionality to objects, such as adding features to a coffee order in a coffee shop system (e.g., adding milk, sugar, or whipped cream).  
  
\*\*Real-World Example:\*\*  
In a coffee shop, you start with a base coffee and add extras like milk, sugar, or caramel. The Decorator pattern allows you to dynamically "decorate" the coffee with these extras.  
  
```python  
class Coffee:  
 def cost(self):  
 return 5  
  
class MilkDecorator:  
 def \_\_init\_\_(self, coffee):  
 self.\_coffee = coffee  
  
 def cost(self):  
 return self.\_coffee.cost() + 1 # Add cost of milk  
  
class SugarDecorator:  
 def \_\_init\_\_(self, coffee):  
 self.\_coffee = coffee  
  
 def cost(self):  
 return self.\_coffee.cost() + 0.5 # Add cost of sugar  
  
# Example usage  
coffee = Coffee()  
coffee\_with\_milk = MilkDecorator(coffee)  
coffee\_with\_milk\_and\_sugar = SugarDecorator(coffee\_with\_milk)  
  
print(coffee\_with\_milk\_and\_sugar.cost()) # 6.5  
```  
  
---  
  
### 5. \*\*Strategy Pattern\*\*  
\*\*Use Case:\*\* Switching between different algorithms or strategies, such as payment methods in an e-commerce application (e.g., credit card, PayPal, or cryptocurrency).  
  
\*\*Real-World Example:\*\*  
An online store allows customers to choose different payment methods. The Strategy pattern lets you encapsulate each payment method and switch between them dynamically.  
  
```python  
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")  
  
class PaymentContext:  
 def \_\_init\_\_(self, strategy):  
 self.\_strategy = strategy  
  
 def set\_strategy(self, strategy):  
 self.\_strategy = strategy  
  
 def pay(self, amount):  
 self.\_strategy.pay(amount)  
  
# Example usage  
context = PaymentContext(CreditCardPayment())  
context.pay(100) # Paid 100 using Credit Card  
  
context.set\_strategy(PayPalPayment())  
context.pay(200) # Paid 200 using PayPal  
```  
  
---  
  
### 6. \*\*Builder Pattern\*\*  
\*\*Use Case:\*\* Constructing complex objects step by step, such as building a custom car or assembling a computer.  
  
\*\*Real-World Example:\*\*  
A car manufacturing company allows customers to customize their cars by choosing the make, model, color, and features. The Builder pattern helps construct the car step by step.  
  
```python  
class Car:  
 def \_\_init\_\_(self):  
 self.make = None  
 self.model = None  
 self.color = None  
  
 def \_\_str\_\_(self):  
 return f"{self.color} {self.make} {self.model}"  
  
class CarBuilder:  
 def \_\_init\_\_(self):  
 self.car = Car()  
  
 def set\_make(self, make):  
 self.car.make = make  
 return self  
  
 def set\_model(self, model):  
 self.car.model = model  
 return self  
  
 def set\_color(self, color):  
 self.car.color = color  
 return self  
  
 def build(self):  
 return self.car  
  
# Example usage  
builder = CarBuilder()  
car = builder.set\_make("Tesla").set\_model("Model S").set\_color("Red").build()  
print(car) # Red Tesla Model S  
```  
  
---  
  
### 7. \*\*Command Pattern\*\*  
\*\*Use Case:\*\* Encapsulating requests as objects, such as implementing an undo/redo system in a text editor.  
  
\*\*Real-World Example:\*\*  
In a text editor, actions like typing, deleting, or formatting text can be encapsulated as commands. This allows the editor to support undo/redo functionality by storing and re-executing commands.  
  
```python  
class Command:  
 def execute(self):  
 pass  
  
 def undo(self):  
 pass  
  
class WriteCommand(Command):  
 def \_\_init\_\_(self, text\_editor, text):  
 self.text\_editor = text\_editor  
 self.text = text  
  
 def execute(self):  
 self.text\_editor.write(self.text)  
  
 def undo(self):  
 self.text\_editor.undo\_write(self.text)  
  
class TextEditor:  
 def \_\_init\_\_(self):  
 self.content = ""  
  
 def write(self, text):  
 self.content += text  
  
 def undo\_write(self, text):  
 self.content = self.content[:-len(text)]  
  
 def show\_content(self):  
 print(self.content)  
  
# Example usage  
editor = TextEditor()  
write\_command = WriteCommand(editor, "Hello, World!")  
  
write\_command.execute()  
editor.show\_content() # Hello, World!  
  
write\_command.undo()  
editor.show\_content() # (empty)  
```  
  
---  
  
These real-world examples demonstrate how design patterns can be applied to solve practical problems in software development.