Getting Started with Design Patterns in Python A Practical Tutorial

By: Waleed Mousa

Design patterns are reusable solutions to common software problems. They provide a way to solve a problem once and reuse the solution in other projects. The concept of design patterns was introduced by the Gang of Four (GoF) in their book "Design Patterns: Elements of Reusable Object-Oriented Software". In this tutorial, we will explore different design patterns and their implementations in Python. Prerequisites

To follow along with this tutorial, you should have a basic understanding of object-oriented programming and Python. It is also recommended that you have some experience with design principles such as SOLID. Table of Contents

#Creational Patterns

Creational patterns are used to create objects in a way that is flexible and provides more control over the object creation process. There are five creational patterns:

1. Singleton Pattern

The Singleton pattern ensures that only one instance of a class is created and provides a global point of access to that instance. The Singleton pattern is useful when we need to ensure that there is only one instance of a class, such as a database connection.

To implement the Singleton pattern in Python, we can use a metaclass. Here's an example:

[ ]:
```
class Singleton(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class MyClass(metaclass=Singleton):
    pass
```

In this example, we define a metaclass called Singleton that keeps track of instances of classes that use it. We then define a class called MyClass that uses the Singleton metaclass. When we create an instance of MyClass, the Singleton metaclass ensures that only one instance of MyClass is created.

2. Factory Pattern

The Factory pattern is used to create objects without specifying the exact class of object that will be created. This pattern is useful when we need to create many objects of different classes that all share a common interface.

To implement the Factory pattern in Python, we can define a class that creates objects based on a string argument. Here's an example:

[ ]:
```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

def get_pet(pet="dog"):
    pets = dict(dog=Dog(), cat=Cat())
    return pets[pet]

d = get_pet("dog")
print(d.speak())
```

```
c = get_pet("cat")
print(c.speak())
```

In this example, we define two classes, Dog and Cat, that both have a method called speak(). We then define a function called get_pet() that returns an instance of either the Dog or Cat class based on the string argument passed to it.

3. Abstract Factory Pattern

The Abstract Factory pattern provides an interface for creating families of related objects without specifying their concrete classes. This pattern is useful when we need to create objects that are related, but not necessarily of the same type.

To implement the Abstract Factory pattern in Python, we can define a factory class that has methods for creating different types of objects. Here's an example:

[ ]:
```
class Dog:
    def speak(self):
        return "Woof!"

class DogFactory:
    def create_pet(self):
        return Dog()

    def create_food(self):
        return "Dog food"

class Cat:
    def speak(self):
        return "Meow!"

class CatFactory:
    def create_pet(self):
        return Cat()

    def create_food(self):
        return "Cat food"

def get_factory(pet="dog"):
    factories = dict(dog=DogFactory(), cat=CatFactory())
    return factories[pet]

factory = get_factory("dog")
pet = factory.create_pet()
print(pet.speak())

food = factory.create_food()
print(food)
```

In this example, we define two classes, Dog and Cat, that both have a method called speak(). We then define two factory classes, DogFactory and CatFactory, that have methods for creating pets and food for their respective animals. We then define a function called get_factory() that returns an instance of either the DogFactory or CatFactory based on the string argument passed to it. Finally, we create an instance of the factory and use it to create a pet and food for a dog.

4. Builder Pattern

The Builder pattern separates the construction of a complex object from its representation, allowing the same construction process to create different representations. This pattern is useful when we need to create objects that have multiple parts and can be constructed in different ways.

To implement the Builder pattern in Python, we can define a class that has methods for building different parts of an object, and a separate director class that uses the builder to construct the object. Here's an example:

[ ]:
```python
class Car:
    def __init__(self):
        self.engine = None
        self.tires = None
        self.gps = None

    def __str__(self):
        return f"{self.engine}, {self.tires}, {self.gps}"

class Builder:
    def build_engine(self):
        pass

    def build_tires(self):
        pass

    def build_gps(self):
        pass

    def get_result(self):
        pass

class Director:
    def __init__(self, builder):
        self.builder = builder

    def construct_car(self):
        self.builder.build_engine()
        self.builder.build_tires()
        self.builder.build_gps()

    def get_car(self):
        return self.builder.get_result()
```

```
class CarBuilder(Builder):
    def __init__(self):
        self.car = Car()

    def build_engine(self):
        self.car.engine = "V8"

    def build_tires(self):
        self.car.tires = "Michelin"

    def build_gps(self):
        self.car.gps = "Garmin"

    def get_result(self):
        return self.car

builder = CarBuilder()
director = Director(builder)
director.construct_car()
car = director.get_car()
print(car)
```

In this example, we define a Car class that has three parts: an engine, tires, and GPS. We then define a Builder class that has methods for building each part, and a Director class that uses the builder to construct the car. Finally, we define a CarBuilder class that implements the Builder interface and provides methods for building each part of the car. We then create an instance of the CarBuilder and use it to construct a car.

5. Prototype Pattern

The Prototype pattern allows us to create new objects by cloning existing ones, rather than creating them from scratch. This pattern is useful when creating new objects is expensive or complex.

To implement the Prototype pattern in Python, we can define a prototype class that has a clone() method that creates a new object with the same properties as the original. Here's an example:

[ ]:
```
import copy

class Prototype:
    def clone(self):
        return copy.deepcopy(self)

class Car(Prototype):
    def __init__(self):
        self.engine = None
        self.tires = None
        self.gps = None
    def __str__(self):
        return f"{self.engine}, {self.tires}, {self.gps}"

car = Car()
car.engine = "V8"
car.tires = "Michelin"
car.gps = "Garmin"

new_car = car.clone()
print(new_car)
```

In this example, we define a Prototype class that has a clone() method that creates a new object using the deepcopy() function from the copy module. We then define a Car class that inherits from Prototype and has three parts: an engine, tires, and GPS. We create an instance of the Car class and set its properties, and then use the clone() method to create a new Car object with the same properties.

#Structural Patterns

Structural patterns are used to organize and structure classes and objects in a way that is flexible and easy to maintain. There are seven structural patterns:

1. Adapter Pattern

The Adapter pattern converts the interface of a class into another interface that clients expect. This pattern is useful when we need to use a class that has a different interface than what our code expects.

To implement the Adapter pattern in Python, we can define a class that adapts the interface of one class to another. Here's an example:

[ ]:
```
class Target:
    def request(self):
        return "Target: The default target's behavior."

class Adaptee:
    def specific_request(self):
        return ".eetpadA eht fo roivaheb laicepS"

class Adapter(Target):
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def request(self):
        return f"Adapter: (TRANSLATED) {self.adaptee.specific_request()[::-1]}"

adaptee = Adaptee()
adapter = Adapter(adaptee)
print(adapter.request())
```

In this example, we define a Target class with a request() method, an Adaptee class with a specific_request() method that returns a string in reverse, and an Adapter class that adapts the interface of Adaptee to match that of Target. We create an instance of Adaptee and an instance of Adapter that uses the Adaptee instance, and then call the request() method on the adapter.

2. Bridge Pattern

The Bridge pattern separates the abstraction from the implementation, allowing them to vary independently. This pattern is useful when we need to decouple an abstraction from its implementation.

To implement the Bridge pattern in Python, we can define an abstraction class and an implementation class, and then use a bridge class to connect them. Here's an example:

[ ]:
```python
class Abstraction:
    def __init__(self, implementation):
        self.implementation = implementation

    def operation(self):
        return f"Abstraction: Base operation with:\n{self.implementation.
 ↪operation_implementation()}"

class Implementation:
    def operation_implementation(self):
        pass

class ConcreteImplementationA(Implementation):
    def operation_implementation(self):
        return "ConcreteImplementationA: Here's the result on the platform A."

class ConcreteImplementationB(Implementation):
    def operation_implementation(self):
        return "ConcreteImplementationB: Here's the result on the platform B."

implementation_a = ConcreteImplementationA()
abstraction = Abstraction(implementation_a)
print(abstraction.operation())

implementation_b = ConcreteImplementationB()
abstraction = Abstraction(implementation_b)
print(abstraction.operation())
```

In this example, we define an Abstraction class and an Implementation class, with the Abstraction class taking an instance of the Implementation class in its constructor. We then define two concrete implementations of the Implementation class, and use them to create two instances of the Abstraction class. Finally, we call the operation() method on each instance to see the different results based on the implementation.

3. Composite Pattern

The Composite pattern allows us to treat a group of objects as a single object, making it easier to work with complex hierarchies. This pattern is useful when we need to work with objects that have a tree-like structure.

To implement the Composite pattern in Python, we can define a Component class that has methods for adding and removing child components, and a Composite class that is a collection of components. Here's an example:

[ ]:
```python
class Component:
    def __init__(self, name):
        self.name = name

    def operation(self):
        pass

    def add(self, component):
        pass

    def remove(self, component):
        pass

class Composite(Component):
    def __init__(self, name):
        super().__init__(name)
        self.children = []

    def operation(self):
        results = []
        for child in self.children:
            results.append(child.operation())
        return f"{self.name}({'+'.join(results)})"

    def add(self, component):
        self.children.append(component)

    def remove(self, component):
        self.children.remove(component)

class Leaf(Component):
    def operation(self):
        return self.name

component = Composite("root")
component.add(Leaf("leaf A"))
component.add(Leaf("leaf B"))

subcomponent = Composite("subtree")
subcomponent.add(Leaf("leaf X"))
subcomponent.add(Leaf("leaf Y"))
```

```
component.add(subcomponent)

print(component.operation())
```

In this example, we define a Component class with methods for adding and removing child components, a Composite class that is a collection of components, and a Leaf class that represents a single component. We create a Composite instance called “root” and add two Leaf instances to it, as well as another Composite instance called “subtree” that has two Leaf instances. Finally, we call the operation() method on the “root” Composite to see the results of all the operations performed on the components.

4. Decorator Pattern

The Decorator pattern allows us to add behavior to an object dynamically, without changing its original structure. This pattern is useful when we need to add functionality to an object without modifying its existing code.

To implement the Decorator pattern in Python, we can define a Component class that has a method for performing a base operation, and a Decorator class that adds additional functionality to the base operation. Here’s an example:

[ ]:
```
class Component:
    def operation(self):
        pass

class ConcreteComponent(Component):
    def operation(self):
        return "ConcreteComponent"

class Decorator(Component):
    def __init__(self, component):
        self.component = component

    def operation(self):
        return self.component.operation()

class ConcreteDecoratorA(Decorator):
    def operation(self):
        return f"ConcreteDecoratorA({self.component.operation()})"

class ConcreteDecoratorB(Decorator):
    def operation(self):
        return f"ConcreteDecoratorB({self.component.operation()})"

component = ConcreteComponent()
decorator_a = ConcreteDecoratorA(component)
decorator_b = ConcreteDecoratorB(decorator_a)
```

```
print(component.operation())
print(decorator_a.operation())
print(decorator_b.operation())
```

In this example, we define a Component class with a method for performing a base operation, a ConcreteComponent class that implements the base operation, a Decorator class that adds additional functionality to the base operation, and two ConcreteDecorator classes that add specific functionality to the base operation. We create an instance of ConcreteComponent and use it to create instances of ConcreteDecoratorA and ConcreteDecoratorB, which add additional functionality to the base operation. Finally, we call the operation() method on each instance to see the results of the different decorators.

5. Facade Pattern

The Facade pattern provides a simplified interface to a complex system, making it easier to use. This pattern is useful when we need to provide a simple interface to a complex set of functionality.

To implement the Facade pattern in Python, we can define a Facade class that has methods for performing complex operations, and then use that class to simplify the interface for clients. Here's an example:

[ ]:
```
class SubsystemA:
    def operation_a1(self):
        pass

    def operation_a2(self):
        pass

class SubsystemB:
    def operation_b1(self):
        pass

    def operation_b2(self):
        pass

class Facade:
    def __init__(self, subsystem_a, subsystem_b):
        self.subsystem_a = subsystem_a
        self.subsystem_b = subsystem_b

    def operation(self):
        results = []
        results.append(self.subsystem_a.operation_a1())
        results.append(self.subsystem_a.operation_a2())
        results.append(self.subsystem_b.operation_b1())
        results.append(self.subsystem_b.operation_b2())
        return results

subsystem_a = SubsystemA()
```

```
subsystem_b = SubsystemB()
facade = Facade(subsystem_a, subsystem_b)
print(facade.operation())
```

In this example, we define two Subsystem classes that have methods for performing complex operations, and a Facade class that has a method for performing all of the complex operations in a simplified way. We create instances of the Subsystem classes and pass them to an instance of the Facade class, which we use to perform the complex operations in a simple way. Finally, we call the operation() method on the Facade instance to see the results of all of the complex operations.

6. Flyweight Pattern

The Flyweight pattern reduces the memory footprint of an object by sharing common data across multiple instances. This pattern is useful when we need to create many objects that share common data.

To implement the Flyweight pattern in Python, we can define a Flyweight class that has a shared state, and a Context class that has an instance of the Flyweight class and its own unique state. Here's an example:

[ ]:
```
class Flyweight:
    def __init__(self, shared_state):
        self.shared_state = shared_state

    def operation(self, unique_state):
        pass

class ConcreteFlyweight(Flyweight):
    def operation(self, unique_state):
        return f"ConcreteFlyweight: ({self.shared_state}, {unique_state})"

class Context:
    def __init__(self, flyweight):
        self.flyweight = flyweight
        self.unique_state = None

    def operation(self, unique_state):
        self.unique_state = unique_state
        return self.flyweight.operation(self.unique_state)

flyweight = ConcreteFlyweight("shared")
context = Context(flyweight)

print(context.operation("unique1"))
print(context.operation("unique2"))
```

In this example, we define a Flyweight class with a shared state, a ConcreteFlyweight class that implements the Flyweight class and has its own unique state, and a Context class that has an instance of the Flyweight class and its own unique state. We create an instance of ConcreteFlyweight with a shared state of "shared", and use it to create an instance of Context. We then call the operation() method on the Context instance twice with different unique states to see the results of the unique states combined with the shared state.

7. Proxy Pattern

The Proxy pattern provides a surrogate or placeholder for another object, allowing us to control access to that object. This pattern is useful when we need to provide controlled access to an object.

To implement the Proxy pattern in Python, we can define a Subject class that has methods for performing an operation, and a Proxy class that acts as a surrogate for the Subject class and controls access to it. Here's an example:

[ ]:
```python
class Subject:
    def operation(self):
        pass

class RealSubject(Subject):
    def operation(self):
        return "RealSubject: Handling request."

class Proxy(Subject):
    def __init__(self, real_subject):
        self.real_subject = real_subject

    def operation(self):
        if self.check_access():
            return self.real_subject.operation()

    def check_access(self):
        return True

real_subject = RealSubject()
proxy = Proxy(real_subject)

print(proxy.operation())
```

In this example, we define a Subject class with a method for performing an operation, a RealSubject class that implements the Subject class and performs the operation, and a Proxy class that acts as a surrogate for the RealSubject class and controls access to it. We create an instance of RealSubject and an instance of Proxy that uses the RealSubject instance, and then call the operation() method on the Proxy instance to see the results of the operation performed by the RealSubject instance. The Proxy class also has a check_access() method that can be used to control access to the RealSubject instance, but in this example it simply returns True to allow access.

#Behavioral Patterns

Behavioral patterns are used to manage algorithms and interactions between objects and classes. There are 11 behavioral patterns:

1. Chain of Responsibility Pattern

The Chain of Responsibility pattern allows us to pass a request through a chain of handlers, each of which can handle the request or pass it on to the next handler in the chain. This pattern is useful when we need to process a request through multiple handlers, without knowing which handler will actually process the request.

To implement the Chain of Responsibility pattern in Python, we can define a Handler class that has a method for handling a request, and a ConcreteHandler class that implements the Handler class and has a reference to the next handler in the chain. Here's an example:

[ ]:
```python
class Handler:
    def set_next(self, handler):
        pass

    def handle(self, request):
        pass

class ConcreteHandlerA(Handler):
    def set_next(self, handler):
        self.next_handler = handler
        return handler

    def handle(self, request):
        if request == "A":
            return f"ConcreteHandlerA: {request}"
        elif self.next_handler:
            return self.next_handler.handle(request)

class ConcreteHandlerB(Handler):
    def set_next(self, handler):
        self.next_handler = handler
        return handler

    def handle(self, request):
        if request == "B":
            return f"ConcreteHandlerB: {request}"
        elif self.next_handler:
            return self.next_handler.handle(request)

handler_a = ConcreteHandlerA()
handler_b = ConcreteHandlerB()

handler_a.set_next(handler_b)

print(handler_a.handle("A"))
print(handler_a.handle("B"))
```

In this example, we define a Handler class with methods for setting the next handler in the chain and handling a request, a ConcreteHandlerA class that implements the Handler class and can handle a request for "A", and a ConcreteHandlerB class that also implements the Handler class and can handle a request for “B”. We create instances of the ConcreteHandler classes and use the set_next() method to set the next handler in the chain. Finally, we call the handle() method on the first handler to see which handler in the chain actually handles the request.

2. Command Pattern

The Command pattern encapsulates a request as an object, allowing us to pass requests as arguments and store them for later use. This pattern is useful when we need to execute requests at different times or in different contexts.

To implement the Command pattern in Python, we can define a Command class that has a method for executing the request, and ConcreteCommand classes that implement the Command class and specify the request to execute. We can also define an Invoker class that has a method for storing and executing commands. Here’s an example:

[ ]:
```
class Command:
    def execute(self):
        pass

class ConcreteCommandA(Command):
    def __init__(self, receiver):
        self.receiver = receiver

    def execute(self):
        return self.receiver.action_a()

class ConcreteCommandB(Command):
    def __init__(self, receiver):
        self.receiver = receiver

    def execute(self):
        return self.receiver.action_b()

class Receiver:
    def action_a(self):
        return "Receiver: Action A"

    def action_b(self):
        return "Receiver: Action B"

class Invoker:
    def set_command(self, command):
        self.command = command

    def execute_command(self):
        return self.command.execute()

receiver = Receiver()
command_a = ConcreteCommandA(receiver)
```

```
command_b = ConcreteCommandB(receiver)

invoker = Invoker()
invoker.set_command(command_a)
print(invoker.execute_command())

invoker.set_command(command_b)
print(invoker.execute_command())
```

In this example, we define a Command class with a method for executing the request, two ConcreteCommand classes that implement the Command class and specify different requests to execute, a Receiver class that has methods for performing the requests, and an Invoker class that has methods for storing and executing commands. We create instances of the Command and Receiver classes, and use the Invoker class to store and execute the commands. Finally, we call the execute_command() method on the Invoker instance to see the results of the commands.

3. Interpreter Pattern

The Interpreter pattern defines a language and a way to interpret that language, allowing us to evaluate expressions and perform other operations. This pattern is useful when we need to parse and interpret complex expressions or languages.

To implement the Interpreter pattern in Python, we can define an AbstractExpression class that has a method for interpreting an expression, and ConcreteExpression classes that implement the AbstractExpression class and interpret specific expressions. Here's an example:

[ ]:
```
class AbstractExpression:
    def interpret(self):
        pass

class TerminalExpression(AbstractExpression):
    def __init__(self, value):
        self.value = value

    def interpret(self):
        return self.value

class NonterminalExpression(AbstractExpression):
    def __init__(self, expressions):
        self.expressions = expressions

    def interpret(self):
        results = []
        for expression in self.expressions:
            results.append(expression.interpret())
        return "+".join(results)

expression = NonterminalExpression([TerminalExpression("A"),␣
 ↪TerminalExpression("B"), TerminalExpression("C")])
```

```
print(expression.interpret())
```

In this example, we define an AbstractExpression class with a method for interpreting an expression, a TerminalExpression class that implements the AbstractExpression class and interprets a single terminal expression, and a NonterminalExpression class that also implements the AbstractExpression class and interprets a series of expressions. We create instances of the TerminalExpression and NonterminalExpression classes, and use them to create an expression that is then interpreted. Finally, we call the interpret() method on the expression to see the results of the interpretation.

4. Iterator Pattern

The Iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. This pattern is useful when we need to traverse a collection of objects without knowing the details of how the collection is implemented.

To implement the Iterator pattern in Python, we can define an Iterator class that has methods for iterating over a collection of objects, and an Aggregate class that has a method for creating an Iterator instance. Here's an example:

[ ]:
```
class Iterator:
    def __init__(self, collection):
        self.collection = collection
        self.index = 0

    def has_next(self):
        return self.index < len(self.collection)

    def next(self):
        item = self.collection[self.index]
        self.index += 1
        return item

class Aggregate:
    def __init__(self):
        self.collection = []

    def add_item(self, item):
        self.collection.append(item)

    def iterator(self):
        return Iterator(self.collection)

aggregate = Aggregate()
aggregate.add_item("A")
aggregate.add_item("B")
aggregate.add_item("C")

iterator = aggregate.iterator()
while iterator.has_next():
    print(iterator.next())
```

In this example, we define an Iterator class with methods for iterating over a collection of objects, and an Aggregate class with a method for creating an Iterator instance. We create an instance of the Aggregate class and add several items to its collection. We then create an instance of the Iterator class using the iterator() method of the Aggregate instance, and use the Iterator instance to iterate over the collection of items and print them to the console.

5. Mediator Pattern

The Mediator pattern provides a way to reduce the coupling between classes by using a mediator to manage the interactions between them. This pattern is useful when we need to manage complex interactions between multiple classes.

To implement the Mediator pattern in Python, we can define a Mediator class that has methods for managing the interactions between classes, and Colleague classes that have a reference to the Mediator class and can send messages to other Colleague instances through the Mediator instance. Here's an example:

[ ]:
```
class Mediator:
    def send(self, message, colleague):
        pass

class ConcreteMediator(Mediator):
    def __init__(self):
        self.colleague_a = None
        self.colleague_b = None

    def set_colleague_a(self, colleague):
        self.colleague_a = colleague

    def set_colleague_b(self, colleague):
        self.colleague_b = colleague

    def send(self, message, colleague):
        if colleague == self.colleague_a:
            self.colleague_b.receive(message)
        else:
            self.colleague_a.receive(message)

class Colleague:
    def __init__(self, mediator):
        self.mediator = mediator

    def send(self, message):
        self.mediator.send(message, self)

    def receive(self, message):
        pass
```

```

class ConcreteColleagueA(Colleague):
    def receive(self, message):
        return f"ConcreteColleagueA received: {message}"

class ConcreteColleagueB(Colleague):
    def receive(self, message):
        return f"ConcreteColleagueB received: {message}"

mediator = ConcreteMediator()
colleague_a = ConcreteColleagueA(mediator)
colleague_b = ConcreteColleagueB(mediator)

mediator.set_colleague_a(colleague_a)
mediator.set_colleague_b(colleague_b)

colleague_a.send("Hello from Colleague A!")
colleague_b.send("Hello from Colleague B!")
```

In this example, we define a Mediator class with methods for managing the interactions between classes, a ConcreteMediator class that implements the Mediator class and manages the interactions between two Colleague classes, and Colleague classes that have a reference to the Mediator class and can send messages to other Colleague instances through the Mediator instance. We create instances of the ConcreteMediator and Colleague classes, and use the ConcreteMediator instance to set the references to the Colleague instances. Finally, we use the send() method on each Colleague instance to send messages to the other Colleague instance, and use the receive() method to see the results of the message passing.

6. Memento Pattern

The Memento pattern provides a way to capture and restore the internal state of an object without violating encapsulation. This pattern is useful when we need to save and restore the state of an object.

To implement the Memento pattern in Python, we can define a Memento class that has a method for getting and setting the state of an object, an Originator class that has a method for creating a Memento instance to capture its internal state, and a Caretaker class that has a method for storing and restoring Memento instances. Here's an example:

[ ]:
```
class Memento:
    def __init__(self, state):
        self.state = state

    def get_state(self):
        return self.state

    def set_state(self, state):
        self.state = state

```

```python
class Originator:
    def __init__(self, state):
        self.state = state

    def create_memento(self):
        return Memento(self.state)

    def restore_memento(self, memento):
        self.state = memento.get_state()

class Caretaker:
    def __init__(self, originator):
        self.mementos = []
        self.originator = originator

    def save_state(self):
        self.mementos.append(self.originator.create_memento())

    def restore_state(self, index):
        self.originator.restore_memento(self.mementos[index])

originator = Originator("Initial state")
caretaker = Caretaker(originator)

caretaker.save_state()

originator.state = "New state"

caretaker.save_state()

print(originator.state)

caretaker.restore_state(0)

print(originator.state)
```

In this example, we define a Memento class with methods for getting and setting the state of an object, an Originator class that has a method for creating a Memento instance to capture its internal state, and a Caretaker class that has methods for storing and restoring Memento instances. We create instances of the Originator and Caretaker classes, and use the Caretaker instance to save the internal state of the Originator instance twice. We then change the internal state of the Originator instance, and use the Caretaker instance to restore the original internal state of the Originator instance by passing the index of the Memento instance to the restore_state() method. Finally, we print the internal state of the Originator instance before and after the restoration.

7. Observer Pattern

The Observer pattern defines a one-to-many dependency between objects, such that when one object changes state, all its dependents are notified and updated automatically. This pattern is useful when we need to maintain consistency between related objects.

To implement the Observer pattern in Python, we can define a Subject class that has methods for managing a list of Observer instances and notifying them of changes to its state, and an Observer class that has a method for receiving notifications of changes to the state of the Subject instance. Here's an example:

[ ]:
```
class Subject:
    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):
        for observer in self.observers:
            observer.update()

class ConcreteSubject(Subject):
    def __init__(self):
        super().__init__()
        self.state = "Initial state"

    def get_state(self):
        return self.state

    def set_state(self, state):
        self.state = state
        self.notify_observers()

class Observer:
    def update(self):
        pass

class ConcreteObserverA(Observer):
    def __init__(self, subject):
        self.subject = subject

    def update(self):
        print(f"ConcreteObserverA received: {self.subject.get_state()}")

class ConcreteObserverB(Observer):
    def __init__(self, subject):
        self.subject = subject
    def update(self):
        print(f"ConcreteObserverB received: {self.subject.get_state()}")

subject = ConcreteSubject()
observer_a = ConcreteObserverA(subject)
observer_b = ConcreteObserverB(subject)

subject.add_observer(observer_a)
subject.add_observer(observer_b)

subject.set_state("New state")
```

In this example, we define a Subject class with methods for managing a list of Observer instances and notifying them of changes to its state, a ConcreteSubject class that extends the Subject class and has methods for getting and setting its internal state, and an Observer class with a method for receiving notifications of changes to the state of the Subject instance. We also define two ConcreteObserver classes that implement the Observer class and receive notifications of changes to the state of the ConcreteSubject instance. We create instances of the ConcreteSubject and ConcreteObserver classes, and use the add_observer() method of the ConcreteSubject instance to add the ConcreteObserver instances to its list of observers. Finally, we use the set_state() method of the ConcreteSubject instance to change its internal state, which triggers the notify_observers() method and notifies the ConcreteObserver instances of the change in state.

8. State Pattern

The State pattern allows an object to change its behavior when its internal state changes. This pattern is useful when we need to change the behavior of an object dynamically based on its internal state.

To implement the State pattern in Python, we can define a State class that has methods for performing actions based on the internal state of an object, and ConcreteState classes that implement the State class and specify the actions to perform for each state. We can also define a Context class that has a reference to a State instance and can change its state dynamically. Here's an example:

[ ]:
```
class State:
    def handle(self):
        pass

class ConcreteStateA(State):
    def handle(self):
        return "ConcreteStateA handles the request"

class ConcreteStateB(State):
    def handle(self):
        return "ConcreteStateB handles the request"

class Context:
    def __init__(self, state):
        self.state = state
```

```python
    def set_state(self, state):
        self.state = state

    def request(self):
        return self.state.handle()

context = Context(ConcreteStateA())
print(context.request())

context.set_state(ConcreteStateB())
print(context.request())
```

In this example, we define a State class with a method for handling requests based on the internal state of an object, two ConcreteState classes that implement the State class and specify the actions to perform for each state, and a Context class that has a reference to a State instance and can change its state dynamically. We create instances of the Context and ConcreteState classes, and use the set_state() method of the Context instance to change its internal state dynamically. Finally, we use the request() method of the Context instance to perform actions based on its internal state and print the results to the console.

9. Strategy Pattern

The Strategy pattern allows us to select an algorithm at runtime, based on the context of a particular operation. This pattern is useful when we need to change the behavior of an object dynamically based on the context of a particular operation.

To implement the Strategy pattern in Python, we can define an AbstractStrategy class that has a method for performing an operation, and ConcreteStrategy classes that implement the AbstractStrategy class and specify the algorithm to use for a particular operation. We can also define a Context class that has a reference to a Strategy instance and can change its behavior dynamically. Here's an example:

[ ]:
```python
class AbstractStrategy:
    def execute(self):
        pass

class ConcreteStrategyA(AbstractStrategy):
    def execute(self):
        return "ConcreteStrategyA executed"

class ConcreteStrategyB(AbstractStrategy):
    def execute(self):
        return "ConcreteStrategyB executed"

class Context:
    def __init__(self, strategy):
        self.strategy = strategy
    def set_strategy(self, strategy):
        self.strategy = strategy

    def execute_strategy(self):
        return self.strategy.execute()

context = Context(ConcreteStrategyA())
print(context.execute_strategy())

context.set_strategy(ConcreteStrategyB())
print(context.execute_strategy())
```

In this example, we define an AbstractStrategy class with a method for performing an operation, two ConcreteStrategy classes that implement the AbstractStrategy class and specify the algorithms to use for a particular operation, and a Context class that has a reference to a Strategy instance and can change its behavior dynamically. We create instances of the Context and ConcreteStrategy classes, and use the set_strategy() method of the Context instance to change its internal strategy dynamically. Finally, we use the execute_strategy() method of the Context instance to perform the selected operation and print the results to the console.

10. Template Method Pattern

The Template Method pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. This pattern is useful when we need to implement an algorithm that follows a common structure but allows for some variability in the steps that are performed.

To implement the Template Method pattern in Python, we can define an AbstractClass with a method that implements the basic structure of the algorithm and calls abstract methods that must be implemented by concrete subclasses. We can also define ConcreteClasses that inherit from the AbstractClass and implement the abstract methods. Here's an example:

[ ]:
```
class AbstractClass:
    def template_method(self):
        self.step_one()
        self.step_two()
        self.step_three()

    def step_one(self):
        pass

    def step_two(self):
        pass

    def step_three(self):
        pass

class ConcreteClassA(AbstractClass):
    def step_one(self):
        return "ConcreteClassA Step 1"
    def step_two(self):
        return "ConcreteClassA Step 2"

class ConcreteClassB(AbstractClass):
    def step_one(self):
        return "ConcreteClassB Step 1"

    def step_two(self):
        return "ConcreteClassB Step 2"

    def step_three(self):
        return "ConcreteClassB Step 3"

a = ConcreteClassA()
print(a.template_method())

b = ConcreteClassB()
print(b.template_method())
```

In this example, we define an AbstractClass with a template_method() that implements the basic structure of the algorithm and calls abstract methods that must be implemented by concrete subclasses. We also define ConcreteClasses that inherit from the AbstractClass and implement the abstract methods. We create instances of the ConcreteClasses and call the template_method() to execute the algorithm with the concrete implementations of the abstract methods.

11. Visitor Pattern

The Visitor pattern allows us to add new operations to existing object structures without modifying those structures. This pattern is useful when we need to perform operations on a complex object structure that cannot be easily modified.

To implement the Visitor pattern in Python, we can define an AbstractVisitor class with methods for each type of object in the structure, and ConcreteVisitor classes that implement the AbstractVisitor class and specify the behavior for each method. We can also define an Element class that has a method for accepting a Visitor instance, and ConcreteElement classes that implement the Element class and specify the behavior for accepting a Visitor instance. Here's an example:

[ ]:
```
class AbstractVisitor:
    def visit_concrete_element_a(self, element):
        pass

    def visit_concrete_element_b(self, element):
        pass

class ConcreteVisitorA(AbstractVisitor):
    def visit_concrete_element_a(self, element):
        return f"ConcreteVisitorA visited {element.name} from ConcreteElementA"
```

```python
    def visit_concrete_element_b(self, element):
        return f"ConcreteVisitorA visited {element.name} from ConcreteElementB"

class ConcreteVisitorB(AbstractVisitor):
    def visit_concrete_element_a(self, element):
        return f"ConcreteVisitorB visited {element.name} from ConcreteElementA"

    def visit_concrete_element_b(self, element):
        return f"ConcreteVisitorB visited {element.name} from ConcreteElementB"

class Element:
    def accept(self, visitor):
        pass

class ConcreteElementA(Element):
    def __init__(self):
        self.name = "ConcreteElementA"

    def accept(self, visitor):
        return visitor.visit_concrete_element_a(self)

class ConcreteElementB(Element):
    def __init__(self):
        self.name = "ConcreteElementB"

    def accept(self, visitor):
        return visitor.visit_concrete_element_b(self)

elements = [ConcreteElementA(), ConcreteElementB()]

visitor_a = ConcreteVisitorA()
visitor_b = ConcreteVisitorB()

for element in elements:
    print(element.accept(visitor_a))
    print(element.accept(visitor_b))
```

In this example, we define an AbstractVisitor class with methods for each type of object in the structure, two ConcreteVisitor classes that implement the AbstractVisitor class and specify the behavior for each method, an Element class with a method for accepting a Visitor instance, and two ConcreteElement classes that implement the Element class and specify the behavior for accepting a Visitor instance.

We create instances of the ConcreteElement classes and add them to a list, and create instances of the ConcreteVisitor classes. Finally, we iterate over the list of ConcreteElement instances and call the accept() method of each instance with the ConcreteVisitor instances, which allows the ConcreteVisitor instances to perform their specified behavior on the ConcreteElement instances without modifying the structure of the objects themselves.

#Finally

In this tutorial, we've covered the most common design patterns in Python, including the Creational, Structural, and Behavioral patterns. These patterns provide a way to organize and structure code in a way that is flexible, reusable, and maintainable. By using design patterns, we can solve common programming problems and make our code more robust and easier to maintain over time.

It's important to note that design patterns are not a silver bullet for every problem, and that it's important to choose the right pattern for the right problem. It's also important to avoid overusing patterns, as this can lead to unnecessarily complex and difficult-to-maintain code.

I hope this tutorial has provided a helpful introduction to design patterns in Python. With practice and experience, you'll become more comfortable using design patterns in your own projects, and you'll find that they can help you write better, more maintainable code.