Python System Design Patterns

A Comprehensive Reference Guide

Introduction

Design patterns are reusable solutions to commonly occurring problems in software design. They represent best practices and provide a template for how to solve a problem in many different situations. This reference covers the essential design patterns used in Python, organized into three main categories: Creational, Structural, and Behavioral patterns.

1. Creational Patterns

Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

1.1 Singleton Pattern

Purpose: Ensures a class has only one instance and provides a global point of access to it.

Use When:

- You need exactly one instance of a class (e.g., database connections, loggers)
- You want to control access to shared resources
- You need global access to an instance

Implementation:

```
class Singleton:
    _instance = None

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

# Usage
s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # True
```

1.2 Factory Pattern

Purpose: Defines an interface for creating objects, but lets subclasses decide which class to instantiate.

Use When:

- Object creation logic is complex
- You want to decouple object creation from usage

- You need to create different objects based on input parameters

Implementation:

```
class Dog:
    def speak(self):
        return 'Woof!'

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

class AnimalFactory:
    @staticmethod
    def create_animal(animal_type):
        if animal_type == 'dog':
            return Dog()
        elif animal_type == 'cat':
            return Cat()
        raise ValueError('Unknown animal type')

# Usage
animal = AnimalFactory.create_animal('dog')
print(animal.speak())  # Woof!
```

1.3 Builder Pattern

Purpose: Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Use When:

- Creating complex objects with many optional parameters
- You want to avoid telescoping constructors
- Object construction requires multiple steps

Implementation:

```
class Car:
    def __init__(self):
        self.make = None
        self.model = None
        self.color = None
        self.engine = None

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 build(self):
        return self.car
# Usage
car = CarBuilder().set_make('Tesla').set_model('Model S').build()
```

1.4 Prototype Pattern

Purpose: Creates new objects by cloning existing objects, avoiding the cost of creating objects from scratch.

Use When:

- Object creation is expensive
- You want to avoid subclassing
- You need to create similar objects with slight variations

Implementation:

```
import copy

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

class Document(Prototype):
    def __init__(self, title, content):
        self.title = title
        self.content = content
# Usage
doc1 = Document('Original', 'Content')
doc2 = doc1.clone()
doc2.title = 'Copy'
```

2. Structural Patterns

Structural patterns deal with object composition, creating relationships between objects to form larger structures.

2.1 Adapter Pattern

Purpose: Converts the interface of a class into another interface that clients expect, allowing incompatible interfaces to work together.

Use When:

- You want to use an existing class with an incompatible interface
- You need to integrate legacy code with new systems
- You want to create reusable classes that cooperate with unrelated classes

Implementation:

```
class EuropeanSocket:
    def voltage(self):
        return 230

class USASocketAdapter:
    def __init__(self, socket):
        self.socket = socket

    def voltage(self):
        return self.socket.voltage() / 2

# Usage
euro_socket = EuropeanSocket()
adapter = USASocketAdapter(euro_socket)
print(adapter.voltage())  # 115
```

2.2 Decorator Pattern

Purpose: Adds new functionality to objects dynamically without altering their structure.

Use When:

- You want to add responsibilities to objects without subclassing
- You need flexible and reusable extensions
- You want to combine multiple behaviors dynamically

Implementation (Python Decorator):

```
def timing_decorator(func):
    import time
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f'Execution time: {end - start}')
        return result
    return wrapper

# Usage
@timing_decorator
def slow_function():
    time.sleep(1)
```

2.3 Facade Pattern

Purpose: Provides a simplified interface to a complex subsystem.

Use When:

- You want to provide a simple interface to a complex system
- You want to layer your subsystems
- You want to reduce dependencies between client code and implementation

Implementation:

```
class CPU:
    def freeze(self): pass
    def execute(self): pass

class Memory:
    def load(self): pass

class HardDrive:
    def read(self): pass

class ComputerFacade:
    def __init__(self):
        self.cpu = CPU()
        self.memory = Memory()
        self.hd = HardDrive()

    def start(self):
        self.cpu.freeze()
        self.memory.load()
        self.hd.read()
        self.cpu.execute()
```

```
# Usage
computer = ComputerFacade()
computer.start()
```

2.4 Proxy Pattern

Purpose: Provides a surrogate or placeholder for another object to control access to it.

Use When:

- You need lazy initialization of expensive objects
- You want to add access control
- You need logging or caching before accessing the real object

Implementation:

```
class RealImage:
    def __init__(self, filename):
        self.filename = filename
        self.load_from_disk()

    def load_from_disk(self):
        print(f'Loading {self.filename}')

    def display(self):
        print(f'Displaying {self.filename}')

class ProxyImage:
    def __init__(self, filename):
        self.filename = filename
        self.real_image = None

    def display(self):
        if self.real_image is None:
            self.real_image = RealImage(self.filename)
        self.real_image.display()

# Usage - loads only when displayed
image = ProxyImage('photo.jpg')
image.display()  # Loading occurs here
```

3. Behavioral Patterns

Behavioral patterns focus on communication between objects, defining how objects interact and distribute responsibility.

3.1 Observer Pattern

Purpose: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified automatically.

Use When:

- Multiple objects need to be updated when one object changes
- You want loose coupling between subject and observers
- You need event-driven systems

Implementation:

```
class Subject:
    def __init__(self):
        self._observers = []

    def attach(self, observer):
        self._observers.append(observer)

    def notify(self, message):
        for observer in self._observers:
            observer.update(message)

class Observer:
    def update(self, message):
        print(f'Received: {message}')

# Usage
subject = Subject()
observer1 = Observer()
subject.attach(observer1)
subject.notify('Event occurred')
```

3.2 Strategy Pattern

Purpose: Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Use When:

- You need different variants of an algorithm
- You want to avoid complex conditional statements
- You want to change algorithms at runtime

Implementation:

```
from abc import ABC, abstractmethod

class PaymentStrategy(ABC):
    @abstractmethod
    def pay(self, amount): pass

class CreditCard(PaymentStrategy):
    def pay(self, amount):
        print(f'Paid ${amount} with credit card')

class PayPal(PaymentStrategy):
    def pay(self, amount):
        print(f'Paid ${amount} with PayPal')

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

    def checkout(self, amount):
        self.strategy.pay(amount)
# Usage
cart = ShoppingCart(CreditCard())
cart.checkout(100)
```

3.3 Command Pattern

Purpose: Encapsulates a request as an object, allowing you to parameterize clients with different requests, queue requests, and support undoable operations.

Use When:

- You want to parameterize objects with operations
- You need to queue, log, or support undo operations
- You want to structure a system around high-level operations

Implementation:

```
class Light:
    def on(self): print('Light is on')
    def off(self): print('Light is off')

class Command(ABC):
    @abstractmethod
    def execute(self): pass

class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light
    def execute(self):
        self.light.on()

class RemoteControl:
    def submit(self, command):
        command.execute()

# Usage
light = Light()
remote = RemoteControl()
remote.submit(LightOnCommand(light))
```

3.4 Iterator Pattern

Purpose: Provides a way to access elements of a collection sequentially without exposing its underlying representation.

Use When:

- You want to access a collection without exposing its internal structure
- You want to support multiple traversal methods
- You want a uniform interface for traversing different structures

Implementation:

```
class BookCollection:
    def __init__(self):
        self._books = []

    def add_book(self, book):
        self._books.append(book)

    def __iter__(self):
        return iter(self._books)

# Usage - Python's built-in iterator
collection = BookCollection()
collection.add_book('Book 1')
for book in collection:
    print(book)
```

3.5 State Pattern

Purpose: Allows an object to alter its behavior when its internal state changes, appearing as if the object changed its class.

Use When:

- Object behavior depends on its state
- You have large conditional statements based on state
- State transitions need to be explicit

Implementation:

```
class State(ABC):
    @abstractmethod
    def handle(self): pass

class OnState(State):
    def handle(self): return 'Device is ON'

class OffState(State):
    def handle(self): return 'Device is OFF'

class Device:
    def __init__(self):
        self.state = OffState()

    def toggle(self):
        if isinstance(self.state, OffState):
            self.state = OnState()
        else:
            self.state = OffState()
        return self.state.handle()

# Usage
device = Device()
print(device.toggle())  # Device is ON
```

4. Python-Specific Best Practices

4.1 Context Manager Pattern

Python's context managers provide a clean way to manage resources using the with statement.

```
class DatabaseConnection:
    def __enter__(self):
        print('Opening connection')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('Closing connection')

# Usage
with DatabaseConnection() as db:
    print('Using database')
```

4.2 Descriptors

Descriptors allow you to customize attribute access in classes.

```
class Validator:
    def __init__(self, min_value, max_value):
        self.min = min_value
        self.max = max_value

    def __set_name__(self, owner, name):
        self.name = name

    def __set__(self, instance, value):
        if not self.min <= value <= self.max:
            raise ValueError(f'{value} not in range')
        instance.__dict__[self.name] = value

class Person:
    age = Validator(0, 120)
```

4.3 Data Classes

Python 3.7+ provides the dataclass decorator for creating classes primarily used to store data.

```
from dataclasses import dataclass
```

```
@dataclass
class Product:
    name: str
    price: float
    quantity: int = 0
```

5. Anti-Patterns to Avoid

- **God Object:** Avoid creating objects that know or do too much
- **Spaghetti Code:** Don't create tangled, hard-to-follow code structures
- **Premature Optimization:** Focus on clarity first, optimize later
- **Cargo Cult Programming:** Understand patterns before applying them
- **Magic Numbers:** Use named constants instead of hardcoded values

6. When to Use Design Patterns

Design patterns are tools, not rules. Use them when they solve a specific problem and make your code more maintainable. Don't force patterns where they don't fit naturally.

Guidelines:

- Start simple and refactor to patterns when complexity emerges
- Choose patterns that match your specific problem
- Consider team familiarity with patterns
- Prioritize readability and maintainability
- Test your implementations thoroughly

End of Reference Guide