**What is the Decorator Design Pattern?**

The **Decorator Design Pattern** allows you to dynamically add or modify the behavior of an object at runtime without altering its structure. It uses composition instead of inheritance and is ideal for scenarios where functionality needs to be extended without modifying the original code.

**2. How does the Decorator pattern differ from inheritance?**

* **Inheritance**: Extends behavior at compile-time by creating a subclass. It’s static and less flexible.
* **Decorator**: Extends behavior at runtime by wrapping objects. It’s dynamic and allows you to apply multiple behaviors to an object flexibly.

**3. Can you explain the concept of wrapping objects with decorators?**

Decorators **wrap an object** to extend its functionality. The decorator class or function has a reference to the original object, delegates calls to it, and adds additional behavior before or after delegating.

Example:

python

Copy code

def bold\_decorator(func):

def wrapper(\*args, \*\*kwargs):

return f"<b>{func(\*args, \*\*kwargs)}</b>"

return wrapper

**4. How are decorators implemented in Python?**

Python decorators are typically implemented as higher-order functions or classes that take a function or class, modify its behavior, and return the modified function or class.

Example of a function decorator:

python

Copy code

def my\_decorator(func):

def wrapper():

print("Before function call")

func()

print("After function call")

return wrapper

@my\_decorator

def say\_hello():

print("Hello!")

**5. What is the difference between a function decorator and a class decorator?**

* **Function decorator**: Modifies or extends the behavior of a function or method.
* **Class decorator**: Modifies or extends the behavior of a class.

Example of a class decorator:

python

Copy code

def add\_str\_method(cls):

cls.\_\_str\_\_ = lambda self: f"Custom string representation of {cls.\_\_name\_\_}"

return cls

@add\_str\_method

class MyClass:

pass

**6. How do you use Python's functools.wraps in a decorator? Why is it important?**

functools.wraps is used to preserve the original function’s metadata (like name and docstring) when wrapping it with a decorator. Without it, the decorated function loses its identity.

Example:

python

Copy code

from functools import wraps

def my\_decorator(func):

@wraps(func)

def wrapper(\*args, \*\*kwargs):

return func(\*args, \*\*kwargs)

return wrapper

**7. How would you chain multiple decorators in Python?**

You can chain multiple decorators by stacking them on top of a function.

Example:

python

Copy code

def bold(func):

def wrapper(\*args, \*\*kwargs):

return f"<b>{func(\*args, \*\*kwargs)}</b>"

return wrapper

def italic(func):

def wrapper(\*args, \*\*kwargs):

return f"<i>{func(\*args, \*\*kwargs)}</i>"

return wrapper

@bold

@italic

def say\_hello():

return "Hello"

print(say\_hello()) # Output: <b><i>Hello</i></b>

**8. Can you give an example where you used a decorator in Python to solve a real-world problem?**

A common use case is **logging**:

python

Copy code

def log\_decorator(func):

def wrapper(\*args, \*\*kwargs):

print(f"Calling {func.\_\_name\_\_} with arguments {args} and {kwargs}")

return func(\*args, \*\*kwargs)

return wrapper

@log\_decorator

def add(a, b):

return a + b

add(2, 3) # Logs the function call and arguments

**9. How would you use a decorator to log the execution time of a function?**

python

Copy code

import time

def time\_logger(func):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

result = func(\*args, \*\*kwargs)

end\_time = time.time()

print(f"{func.\_\_name\_\_} executed in {end\_time - start\_time:.2f} seconds")

return result

return wrapper

@time\_logger

def slow\_function():

time.sleep(2)

slow\_function()

**10. Write a decorator that limits the number of times a function can be called.**

python

Copy code

def limit\_calls(max\_calls):

def decorator(func):

calls = 0

def wrapper(\*args, \*\*kwargs):

nonlocal calls

if calls >= max\_calls:

raise RuntimeError(f"{func.\_\_name\_\_} exceeded {max\_calls} calls")

calls += 1

return func(\*args, \*\*kwargs)

return wrapper

return decorator

@limit\_calls(3)

def greet():

print("Hello!")

greet()

greet()

greet()

# greet() -> Raises RuntimeError

**11. What are some potential pitfalls of using decorators?**

* May obscure the original function’s logic, especially with nested or complex decorators.
* Without functools.wraps, debugging can be harder as metadata is lost.
* Overusing them can reduce code readability.
* Managing side effects introduced by decorators can be tricky.

**12. How can you pass arguments to a decorator?**

By wrapping the decorator in another function that takes arguments:

python

Copy code

def repeat(times):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

for \_ in range(times):

func(\*args, \*\*kwargs)

return wrapper

return decorator

@repeat(3)

def say\_hi():

print("Hi!")

say\_hi()

**13. How can the Decorator pattern improve code reusability?**

Decorators encapsulate common behavior (e.g., logging, authorization, caching) in reusable, modular components that can be applied to multiple objects without duplicating code.

**14. Can decorators be used for class methods? How?**

Yes, decorators can be applied to class methods like any other function. Use @classmethod or @staticmethod decorators where necessary.

Example:

python

Copy code

def log\_method(func):

def wrapper(self, \*args, \*\*kwargs):

print(f"Calling {func.\_\_name\_\_} from {self.\_\_class\_\_.\_\_name\_\_}")

return func(self, \*args, \*\*kwargs)

return wrapper

class MyClass:

@log\_method

def greet(self):

print("Hello!")

obj = MyClass()

obj.greet()

**15. How is the Decorator pattern in Python different from Java or C++ implementations?**

* **Python**: Leverages first-class functions, making it simpler and more natural to apply decorators.
* **Java/C++**: Requires explicit class hierarchies and often verbose code to implement decorators.

**16. When would you avoid using decorators and prefer alternative approaches?**

* If the logic is too complex and reduces readability.
* When there’s a simpler solution with fewer abstraction layers.
* When behavior depends on global state, which can be hard to debug in decorators.

**17. Compare the Decorator pattern with the Proxy pattern.**

| **Feature** | **Decorator** | **Proxy** |
| --- | --- | --- |
| Purpose | Adds functionality dynamically | Controls access to an object |
| Structure | Wraps objects for additional behavior | Acts as a placeholder for an object |
| Flexibility | Extends behavior | Focused on managing access |