ChatGPT

Decorator Design Pattern in Python

Introduction

The **Decorator Design Pattern** allows behavior to be added to individual objects or functions dynamically, without modifying their original code. In simple terms, a decorator "wraps" another function or object and intercepts calls to add new functionality before or after delegating to the original. In Python, a decorator is typically a function (or class) that takes another function as an argument and returns a new function with enhanced behavior [1]. This is a powerful feature for writing **clean, reusable code** because it lets you add features (like logging, authentication, caching, etc.) without changing the original function's source code [2]. The following sections present a progressive tutorial, from basic to advanced, on how to implement and use decorators in Python.

Beginner Level – Function Decorators Basics

Concept & Intent: At the beginner level, a decorator can be understood as a wrapper around a function. Imagine a plain gift box (the original function) that you wrap with decorative paper and ribbons (the added functionality) – the gift inside remains the same, but its presentation is enhanced. Similarly, a function decorator allows you to add code **before and/or after** a function runs, without altering the function's own code. This is done by defining an *outer function* (the decorator) that returns an *inner function* (the wrapper) which calls the original function. The decorator can then be applied to any target function using the `@decorator_name` syntax placed above the function definition.

Python Function Decorator Example: Below is a simple function decorator that adds extra behavior before and after a greeting function. The example uses an analogy of announcing actions before and after the core action (saying hello), illustrating the wrapping concept:

```
def my_decorator(func):
    """A simple decorator that wraps a function to print messages before and
after its call."""
    def wrapper():
        # Behavior added before the original function call
        print("Preparing to say hello...")
        result = func()  # call the original function
        # Behavior added after the original function call
        print("Finished saying hello.")
        return result  # return the original function's result (if any)
    return wrapper

@my_decorator
def say_hello():
    """Original function that simply prints a greeting."""
    print("Hello!")
```

```
# Using the decorated function
say_hello()
```

Output:

```
Preparing to say hello...
Hello!
Finished saying hello.
```

How it works: In the code above, `my_decorator` is a function that takes `func` (the original function) and defines an inner function `wrapper()` that wraps the call to `func` with additional print statements. The line `@my_decorator` above `say_hello` is syntactic sugar, equivalent to `say_hello = my_decorator(say_hello)` [3]. When we call `say_hello()`, we are actually calling the `wrapper()` function inside `my_decorator`. The wrapper executes the extra code (printing messages) before and after calling the original `say_hello` function. Importantly, the original function's behavior ("Hello!" message) is preserved and executed in order, but now it's decorated with new behavior around it.

Pattern Fit: This function decorator demonstrates the core decorator pattern principle: it **adds new behavior** (printing messages) to an existing function dynamically, without modifying the function's internal code. We achieved this by wrapping the function call in a new function. You could wrap (decorate) the same `say_hello` function with *different* decorators for different behaviors, or apply `my_decorator` to any other function, showing how decorators promote code reuse and extension.

Use Cases (Beginner): Function decorators are handy for many common tasks: - **Logging or Tracing:** Automatically print logs whenever a function is called (inputs, outputs, etc.). - **Timing:** Measure and print the execution time of functions. - **Access Control/Validation:** Check prerequisites (like user authentication or input validation) before running a function. - **Resource Management:** Acquire and release resources around function calls (e.g., open and close a file, manage database transactions).

Instead of writing these cross-cutting concerns inside every function, you can write a decorator once and apply it wherever needed. This keeps the **original functions clean and focused** on their primary job (following the *Single Responsibility Principle*).

Best Practices (Beginner): When writing simple decorators, ensure that your wrapper function: - Calls the original function (`func`) with all its arguments and returns its result (to not break the expected behavior). - Preserves the original function's metadata (name, docstring) if needed. (In advanced usage, Python provides `functools.wraps` to help with this; more on that later.) - Keeps the added behavior minimal and generic if you intend to reuse the decorator across many functions.

With a solid grasp of basic function decorators, we can move on to more advanced techniques, such as using classes to implement decorators for additional flexibility.

Intermediate Level – Class-Based Decorators

Concept & Intent: Python also allows creating decorators using classes. A **class-based decorator** is simply a class that implements the `__call__` method, making its instances callable like functions [4]. Class-based decorators provide greater flexibility for more complex scenarios, especially when you need

to maintain **state** between function calls or when organizing code in an object-oriented way. Instead of an inner function closure to carry state, you can use instance attributes. This can make complex decorators easier to understand and extend.

For example, suppose we want a decorator that counts how many times a function has been called – a stateful behavior that persists across calls. This is cumbersome with a simple function decorator (it would require a mutable closure or global variable), but straightforward with a class. Below is an example of a class-based decorator that counts calls:

```
class CallCounter:
    """A decorator class that counts the number of times a function is
called."""
    def __init__(self, function):
        self.function = function
        self.count = 0  # initialize call count

    def __call__(self, *args, **kwargs):
        # This method is called when the decorated function is invoked.
        self.count += 1
        print(f"Function '{self.function.__name__}' has been called
{self.count} times.")
        # Forward the call to the original function and return its result
        return self.function(*args, **kwargs)

@CallCounter
def say_hello(name):
    print(f"Hello, {name}!")

# Using the decorated function:
say_hello("Alice")
say_hello("Bob")
say_hello("Charlie")
```

Output: (The exact order of lines may differ because each call prints a count then the greeting)

```
Function 'say_hello' has been called 1 times.
Hello, Alice!
Function 'say_hello' has been called 2 times.
Hello, Bob!
Function 'say_hello' has been called 3 times.
Hello, Charlie!
```

How it works: When `@CallCounter` decorates `say_hello`, Python does the following under the hood: `say_hello = CallCounter(say_hello)`. Here `CallCounter(say_hello)` invokes the class's `__init__`, storing the original function in `self.function`. The resulting `say_hello` is now an instance of `CallCounter`. When you call `say_hello("Alice")`, Python actually calls the `CallCounter.__call__` method. Inside `__call__`, we increment the counter and then delegate to the original function (`self.function`) with the given arguments. The state (`self.count`) persists

in the `CallCounter` object between calls, allowing the decorator to remember how many times the function has been invoked. This example clearly shows a case where a class-based decorator is beneficial: maintaining a counter without global variables.

Advantages of Class-Based Decorators:
- **Stateful Decorators:** Because the decorator is an object, you can easily maintain state in instance attributes (as shown with `self.count`) [5]. This avoids using global variables or closures for tracking state across calls. Other examples might include caching results or keeping track of execution times across calls.
- **Clarity and Organization:** For complex decorators, encapsulating the logic in a class (with multiple methods or helper functions) can improve readability. The intent can be clearer when the decorator's behavior is grouped in a class rather than hidden in nested function scopes.
- **Reusability:** Class decorators can be extended via inheritance if needed, or combined with other object-oriented features. They also naturally lend themselves to configuration by passing parameters to the class constructor (more on parameterized decorators in the next sections).

Use Cases (Intermediate): Use class-based decorators when you need more than what a simple function closure can easily provide: - **Maintaining State:** Counting calls, caching/memoization (storing results in the instance), throttling (allowing function to run only every N seconds by storing last run time), etc. - **Complex Setup/Teardown:** If your decorator needs elaborate setup, a class's initializer can handle it cleanly. - **Configurable Decorators:** By using a class, you can accept arguments when creating the decorator (by implementing an `__init__` that takes extra parameters besides the function).

For instance, you might create a decorator class `Retry` that on each call tries to run the function and catches exceptions, retrying a certain number of times. The number of retries could be provided when constructing the decorator (`@Retry(max_attempts=3)`), stored as an instance attribute, and used in the `__call__` logic.

Best Practices (Intermediate): When writing class-based decorators: - Remember that the decorator instance replaces the function, so implement `__call__` to handle all arguments and return the function's result (just like a normal wrapper function). - Use `functools.update_wrapper` or `functools.wraps` inside `__call__` if you want the decorated object to mimic the original function's metadata. For example, you can call `functools.update_wrapper(self, function)` in the constructor to copy over the function's name, docstring, etc., onto the instance. This is a bit advanced, but it ensures tools like `help()` or introspection see the decorator-wrapped function as the original.
- Keep the class focused on one responsibility. If it's doing too many things, consider splitting into multiple decorators or helper functions for clarity.

Now that we understand function and class decorators, let's integrate decorators with broader object-oriented programming concepts, exploring the classic structural **Decorator Pattern** as it applies to objects and classes.

Advanced Level – Decorators in OOP (Structural Pattern Example)

Concept & Intent: In classical design pattern terminology, the **Decorator is a structural pattern** that allows you to attach new behaviors to objects at runtime by placing them inside wrapper objects that contain those behaviors [6]. In other words, rather than modifying or subclassing a class to add features, you wrap an object with a *decorator object* that implements the same interface and adds its own behavior. This is often used to adhere to the *Open/Closed Principle* (classes are open for extension

but closed for modification). Each decorator object holds a reference to the original (or another decorator) and delegates operations to it, adding something extra either **before or after** delegating.

Real-World Analogy: Consider a basic car that can drive. If you want to add features like bulletproof armor or turbo boost, you could create new specialized car subclasses for each combination (ArmoredCar, TurboCar, ArmoredTurboCar, etc.), but that leads to an explosion of classes. Instead, using the decorator pattern, you could have the basic Car and then wrap it with an ArmorDecorator or TurboDecorator at runtime. The ArmorDecorator knows how to handle damage (added behavior) but for driving it defers to the underlying car. Multiple decorators can be stacked (e.g., a turbo car that is also armored) without modifying the original Car class or creating a new class for every combination.

Python Example – Text Formatting using Decorator Classes:
Let's demonstrate the structural decorator pattern with a simple text formatting example. We have a core component class that provides plain text, and we have several decorator classes (Bold, Italic, Underline) that wrap a text component to add formatting. Each decorator will implement the same interface (`render()` method in this case) as the component it wraps.

```
class Text:
    """Core component class that simply stores text."""
    def __init__(self, content: str):
        self.content = content
    def render(self) -> str:
        """Return the text content (no formatting)."""
        return self.content

class BoldDecorator(Text):
    """Decorator that wraps a Text component to add <b> (bold) formatting."""
    def __init__(self, wrapped: Text):
        self.wrapped = wrapped  # store the component to decorate
    def render(self) -> str:
        # Call the wrapped component's render and add bold HTML tags around
it
        return f"<b>{self.wrapped.render()}</b>"

class ItalicDecorator(Text):
    """Decorator that wraps a Text component to add <i> (italic)
formatting."""
    def __init__(self, wrapped: Text):
        self.wrapped = wrapped
    def render(self) -> str:
        return f"<i>{self.wrapped.render()}</i>"

class UnderlineDecorator(Text):
    """Decorator that wraps a Text component to add <u> (underline)
formatting."""
    def __init__(self, wrapped: Text):
        self.wrapped = wrapped
    def render(self) -> str:
        return f"<u>{self.wrapped.render()}</u>"
```

```
# Usage of the decorators:
plain_text = Text("Hello, World")
# Wrap the text with multiple decorators
decorated_text = BoldDecorator( ItalicDecorator(
UnderlineDecorator(plain_text) ) )
print("Original:", plain_text.render())       # Original text
print("Decorated:", decorated_text.render())  # Text with
bold+italic+underline
```

Output:

```
Original: Hello, World
Decorated: <b><i><u>Hello, World</u></i></b>
```

How it works: The class `Text` is our base component providing a `render()` method. The decorator classes `BoldDecorator`, `ItalicDecorator`, and `UnderlineDecorator` each take a `Text` (or already-decorated Text) object and also provide a `render()` method. However, instead of producing text on their own, they **delegate** the call to the wrapped object's `render()` and then enhance the result. For example, `UnderlineDecorator.render()` calls its wrapped object's `render()` (which could be a plain `Text` or another decorator), and then wraps the returned string with `<u>...</u>` tags to underline it. Each decorator follows the same interface (`render()`) as `Text`, so they can be used interchangeably wherever a `Text` is expected.

In the usage, we start with a `Text("Hello, World")` object. We then create an `UnderlineDecorator(plain_text)`, which produces an underlined text component. We pass that to `ItalicDecorator` to italicize the underlined text, and then to `BoldDecorator` to bold the italic-underlined text. Finally, calling `decorated_text.render()` triggers a chain of calls: Bold -> Italic -> Underline -> Text, each adding its formatting. The result is the original string wrapped in all specified HTML tags. Notice how we achieved a combination of behaviors (underline + italic + bold) by layering single-responsibility decorators, rather than needing a single class that knows about all three.

Pattern Fit: This example illustrates the **decorator design pattern** in its pure form. We dynamically attached new behaviors to the object `plain_text` by wrapping it in decorator objects, without altering `Text` or creating subclasses of `Text` for every feature combination. Each decorator is **modular** – you could add more decorators (e.g., a ColorDecorator to add color) and mix and match them as needed. The wrapped object and the decorators all share the same interface (`render()`), which makes the composition seamless and transparent to the client using `render()`. In fact, the client can treat `decorated_text` as if it were just a `Text` object.

Use Cases (Advanced OOP): The structural decorator pattern is useful in many OOP scenarios: - **GUI Components:** For example, in GUI frameworks, you might decorate a basic window with scrollbars, borders, or other features. Each feature can be a decorator that adds functionality (scrolling, visual border) while delegating core behavior to the underlying window component. - **Streams and I/O:** As noted in many design pattern discussions, stream libraries often use decorators (e.g., a basic file stream decorated with buffering, compression, or encryption streams) [7]. Each wrapper adds functionality (like buffering or encrypting data) on top of the underlying stream. - **Game or Simulation Objects:** Add abilities to game characters or entities by wrapping them (e.g., an `AttackPowerUpDecorator` that gives an object extra attack capability without altering the object class). - **Formatting and Data**

Transformation: As shown with text, you can layer transformations (like parsing, filtering, formatting) by stacking decorators around data.

Best Practices (Advanced/OOP):
- Ensure that your base component and decorators share a common interface (either via inheritance or simply by convention). This allows the decorators to be transparent replacements for the base object. - Each decorator should focus on a single enhancement. This keeps them reusable and combinable. If you find a decorator doing too many things, consider splitting it. - Be mindful of the order of wrapping. Different order can yield different results, especially if the operations are not commutative (e.g., formatting tags or in stream processing). - Because multiple layers can become complex, it's good to document or name decorators clearly (e.g., `EncryptedBufferedFileStream` might be easier to reason about than applying `EncryptedStream(BufferedStream(FileStream))` on the fly – although the latter is more flexible). - In Python specifically, this pattern is often easier to implement using function decorators for simple cases. However, understanding the object-oriented decorator pattern helps in structuring more complex systems and in recognizing such patterns in libraries.

Having explored function decorators and the decorator pattern with classes, we can move to professional-level insights: how decorators are used in real-world applications, how to combine them, and best practices for writing robust decorators.

Professional Level – Advanced Patterns and Best Practices

Concept & Intent: In professional Python development, decorators are heavily used to handle **cross-cutting concerns** – aspects of a program that are needed across many functions or methods, such as logging, authentication, caching, error handling, and performance timing. By encapsulating these in decorators, you keep the core business logic clean and separately manage these auxiliary concerns. At this level, we focus on writing **modular, extensible decorators** and using multiple decorators in combination for sophisticated behavior. We'll also cover important best practices that seasoned developers follow when using decorators.

Multiple Decorators (Stacking): Python allows you to apply more than one decorator to a function by stacking `@decorator` lines above the function. The decorators will wrap the function one after another in the order they appear (with the one closest to the function applying first). For example, if we have a logging decorator and an authentication decorator, we could stack them:

```
def log_calls(func):
    """Decorator to log function call details (arguments and return)."""
    import functools
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print(f"[LOG] Calling {func.__name__} with args={args},
kwargs={kwargs}")
        result = func(*args, **kwargs)
        print(f"[LOG] {func.__name__} returned {result}")
        return result
    return wrapper

def require_auth(func):
    """Decorator to enforce authentication (expects first arg to have
```

```
'is_admin' attribute)."""
    import functools
    @functools.wraps(func)
    def wrapper(user, *args, **kwargs):
        # Simple check: user must be admin to proceed
        if not getattr(user, "is_admin", False):
            raise PermissionError("User lacks permissions to execute this
function.")
        return func(user, *args, **kwargs)  # user is authorized, proceed
    return wrapper

# Example usage of stacking decorators:
@log_calls
@require_auth
def delete_user(user, user_id):
    # Core logic to delete a user by ID
    # (In practice, this might delete from a database)
    return f"User {user_id} deleted."

# Simulate a user object:
class User:
    def __init__(self, name, is_admin):
        self.name = name
        self.is_admin = is_admin

admin = User("Alice", is_admin=True)
guest = User("Bob", is_admin=False)

print("Admin trying to delete user 42:")
print(delete_user(admin, 42))   # This should succeed and be logged

print("\nGuest trying to delete user 43:")
print(delete_user(guest, 43))   # This should raise an error due to
require_auth
```

Expected Output:

```
Admin trying to delete user 42:
[LOG] Calling delete_user with args=(<User object at 0x...>, 42), kwargs={}
[LOG] delete_user returned User 42 deleted.
User 42 deleted.

Guest trying to delete user 43:
[LOG] Calling delete_user with args=(<User object at 0x...>, 43), kwargs={}
Traceback (most recent call last):
  ...
PermissionError: User lacks permissions to execute this function.
```

In this example, `@require_auth` ensures the user has proper rights before executing `delete_user`, and `@log_calls` records the call and result. The order of stacking is important: here `log_calls` is above `require_auth`, so `log_calls` wraps the function *after* it has been authenticated. This means the log will still record a call even if the user is unauthorized (since the `require_auth` wrapper will raise an error *inside* the logging wrapper). If we swapped the order (`@require_auth` on top of `@log_calls`), the authentication check would happen first, potentially preventing the logging from even running for unauthorized calls. This illustrates how stacking can influence behavior, and you should choose the order deliberately based on the desired outcome [8].

Real-World Use Cases: Decorators shine in real-world applications and frameworks. Some typical scenarios include:
- **Logging and Debugging:** Using decorators to log function entry/exit, arguments, and return values for debugging or audit trails [9]. For example, a `@debug` decorator might print detailed info each time any function is called. Logging decorators help centralize and standardize logging logic.
- **Authentication and Authorization:** In web frameworks like Flask or Django, decorators are used to enforce access controls. A common example is `@login_required` in Django or Flask, which checks if a user is logged in (and possibly has the right role) before allowing the view function to run [10]. This keeps authentication checks out of the core view logic.
- **Caching/Memoization:** Expensive computations can be decorated to cache their results. Python's standard library provides `@functools.lru_cache` for functions, which is a decorator that caches return values so that subsequent calls with the same arguments are fast [11]. In your own code, you could write a `@cache_result` decorator that stores results in a dictionary for quick lookup, as long as the function's inputs are hashable. This can drastically improve performance for pure functions or idempotent methods.
- **Timing and Profiling:** A `@timeit` decorator can record how long a function takes to execute each time, which is useful for performance analysis. This might log or accumulate timing stats without cluttering the function itself with timing code.
- **Retry and Error Handling:** In network or file operations, a `@retry` decorator can automatically catch exceptions and retry the function a certain number of times or with delays. This makes the function appear reliable to callers while encapsulating the retry logic in one place.
- **Validation and Sanitization:** You can use decorators to validate inputs (e.g., types, ranges, non-null) or sanitize data before it reaches the main logic. For example, a `@validate_json` decorator in a web API can ensure the request data is correctly formatted and all required fields are present.
- **Event Handling / Registration:** Some frameworks use decorators to register functions for certain events or routes. For instance, Flask uses `@app.route('/path')` to register a function as a web route handler. This decorator doesn't modify the function's behavior per se, but it attaches metadata to the function (or registers it in some global registry) so that the framework knows about it. This is an alternate use of decorators for declarative programming.

By leveraging decorators in these scenarios, you achieve separation of concerns: the function's primary job is written clearly, and the ancillary tasks (logging, security checks, etc.) are handled by decorators.

Best Practices (Professional): When developing or using decorators in a large codebase, keep the following best practices in mind: - **Use `functools.wraps`:** Always apply `@functools.wraps(func)` to your inner wrapper function (in function decorators) or use `functools.update_wrapper` for class decorators. This preserves the original function's metadata (`__name__`, `__doc__`, etc.), which is crucial for debugging and introspection. Without this, your decorated function might have a generic name like `"wrapper"` and no docstring, which can be confusing. The `functools.wraps` decorator handles copying these attributes for you [12] [13].
- **Documentation:** Clearly document what your decorator does, especially any expectations on the

function (e.g., our `require_auth` assumes the first argument has `is_admin` attribute). Users of the decorator should know its effect and requirements without reading its internals.
- **Avoid Changing the Core Signature:** A well-behaved decorator should not arbitrarily change the calling signature or return type of the function. It should ideally be completely transparent other than the added side effect. There are cases (like adding parameters) where this rule is bent, but in general, aim for decorators that *extend* behavior, not fundamentally change what the function is supposed to do.
- **Order of Decoration:** As discussed, the stacking order matters. If multiple decorators are applied, consider how they interact. A good practice is to have each decorator focus on a separate concern and try to make them order-agnostic if possible, or clearly document if one should be applied before another.
- **Don't Overuse:** While decorators are powerful, over-decorating can make code hard to follow. If you stack many decorators, someone reading the code has to mentally unravel multiple layers to understand what happens when the function is called. Use them judiciously and ensure the benefit (reducing repetition, enforcing a policy, etc.) outweighs the added indirection.
- **Test Decorators in Isolation:** Because decorators can affect multiple functions, it's wise to test them separately. For example, if you write a caching decorator, create test functions to ensure it caches correctly. Also test the decorated functions to ensure they still behave as expected with the decorator applied.
- **Composable and Configurable:** Aim to write decorators that are *composable* (can work with other decorators without conflict) and *configurable* when appropriate. For instance, you can write a parameterized decorator (one that takes arguments itself) to allow flexibility. A common pattern for that is an extra outer function. For example:

```
def repeat(num_times):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(num_times=3)
def greet(name):
    print(f"Hello {name}")
```

Here `@repeat(3)` will cause `greet` to execute 3 times on each call. The ability to pass an argument (`num_times`) makes the decorator more reusable. Many real-world decorators, like `@timeout(seconds=5)` or `@permission(required_role="admin")`, use this pattern of a decorator factory.

In summary, at a professional level you treat decorators as first-class tools for building maintainable and scalable systems. They help implement **concerns that cut across many parts of an application** (security, logging, etc.) in one place. Decorators should make code *more readable* by pulling out repetitive logic, and with careful use of the best practices above, they will do so without introducing confusion.

Conclusion

We have progressively explored the Decorator Design Pattern in Python, from simple function decorators for beginners to advanced usage in object-oriented designs and real-world scenarios. To recap: - At the **Beginner** level, we learned what decorators are and created basic function decorators, using analogies and simple examples to see how they wrap functionality. - At the **Intermediate** level, we introduced class-based decorators, which offer statefulness and clarity for more complex behaviors. - At the **Advanced** level, we connected the concept back to the formal **Decorator Pattern** in OOP, demonstrating how decorators can be used to dynamically extend object behavior while keeping code modular and adhering to SOLID principles. - At the **Professional** level, we looked at practical patterns like logging, authentication, and caching, and emphasized best practices (like using `functools.wraps` and mindful stacking) that ensure our decorators remain robust and maintainable.

By building on each layer of knowledge, you should now have a solid understanding of how to implement and use decorators in Python effectively. Decorators are a powerful asset in a Pythonista's toolkit – they enable cleaner code, reuse of common functionality, and adherence to design principles that keep code flexible and extensible [14] [15]. As you apply these patterns, always remember the guiding intent: **extend functionality without modifying existing code**, achieving new capabilities while keeping the original codebase stable and uncluttered. Happy decorating!

---

[1] Python Decorators (With Examples)
https://www.programiz.com/python-programming/decorator

[2] [3] 10 Advanced Use Cases for Decorators in Python
https://developer-service.blog/10-advanced-use-cases-for-decorators-in-python/

[4] [5] [10] [11] How to Use Python Decorators (With Function and Class-Based Examples) | DataCamp
https://www.datacamp.com/tutorial/decorators-python

[6] Decorator Method – Python Design Patterns | GeeksforGeeks
https://www.geeksforgeeks.org/decorator-method-python-design-patterns/

[7] Decorator in Python / Design Patterns
https://refactoring.guru/design-patterns/decorator/python/example

[8] [9] [12] [13] [14] [15] Primer on Python Decorators – Real Python
https://realpython.com/primer-on-python-decorators/