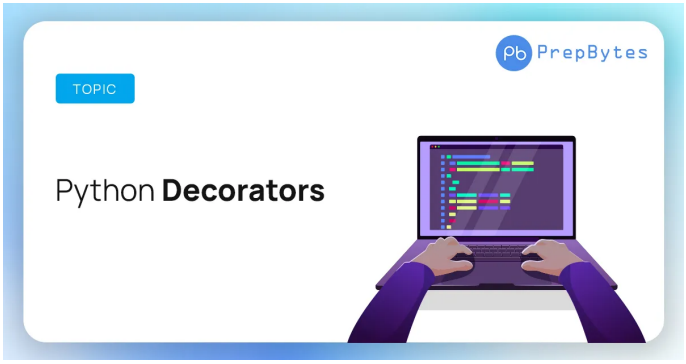
“Mastering Python Decorators: A Comprehensive Guide for Enhancing Code Modularity and Functionality”

Ewho Ruth · Follow
122 min read · May 25



1. Introduction

1.1 What are decorators?
Decorators are a powerful feature in Python that allow the modification or enhancement of functions, classes, or other objects at runtime without directly modifying their source code. Decorators are implemented using functions or classes and are used to wrap or decorate the target object with additional functionality.

2 Why decorators are important in Python

Decorators play a crucial role in Python programming for several reasons:

a) **Code Reusability**: Decorators enable the separation of cross-cutting concerns, such as logging, validation, authentication, or caching, from the core logic of functions or classes. This promotes code reuse and modularity.

b) **Readability and Maintainability**: Decorators provide a clean and concise way to add functionality to existing code without cluttering the original implementation. They improve code readability by isolating specific behaviors and making code easier to understand and maintain.

c) **Meta-programming and Extensibility**: Decorators allow for dynamic modification of code at runtime, enabling advanced meta-programming techniques. They provide a flexible mechanism for extending and customizing the behavior of functions or classes without modifying their original definition.

d) **Frameworks and Libraries**: Decorators are widely used in Python frameworks and libraries, such as Flask, Django, and SQLAlchemy. They provide a standardized way to enhance and customize the behavior of these frameworks, making it easier to develop complex applications.

e) **Language Constructs**: Python itself utilizes decorators for built-in language features, such as property getters and setters, class methods, static methods, and abstract methods. Understanding decorators is essential for leveraging these language constructs effectively.

In summary, decorators are a fundamental part of Python’s expressive and dynamic nature. They empower developers to write cleaner, more modular, and extensible code, and they are a key tool in building and extending Python frameworks and libraries.

2.1 **Understanding Function Decorators**

Function decorators in Python allow you to modify the behavior of a function without directly modifying its source code. They are implemented using higher-order functions or classes. When a function is decorated, it is passed as an argument to the decorator and the decorator returns a modified version of the function.

2.2 **Decorator syntax and usage**: Decorators are implemented using the “@” symbol followed by the name of the decorator function or class. They are placed before the definition of the function to be decorated. The decorator function/class is responsible for modifying or enhancing the behavior of the target function. Here’s an example:

```
@decorator
def my_function():
```

```
    # Function code
```

2.3 **Decorating Functions with Arguments**

Decorators can accept arguments themselves, allowing you to customize the behavior of the decorated function. To achieve this, you can create a decorator factory function that takes arguments and returns the decorator function. The decorator function can then wrap the original function and apply the desired behavior based on the arguments.

Example:

```
def decorator_factory(argument):
    def decorator(func):
        def wrapper(*args, **kwargs):
            # Pre-decoration logic
            print("Decorator argument:", argument)
            result = func(*args, **kwargs)
            # Post-decoration logic
            return result
        return wrapper
    return decorator

@decorator_factory("example_argument")
def my_function():
    # Function code
    print("Inside my_function")

# Test the decorated function
my_function()
```

Explanation:

1. The `decorator_factory` is a function that takes an argument and returns a decorator function.
2. The decorator function takes a target function as an argument and returns a wrapper function.
3. The wrapper function performs the pre-decoration logic, such as printing the decorator argument.
4. The wrapper then calls the original function (`func`) with the provided arguments and stores the result.
5. Finally, the wrapper performs the post-decoration logic and returns the result.

`my_function` is decorated with the `decorator_factory("example_argument")` decorator. This means that the `decorator_factory` function is called with the argument `"example_argument"` , and it returns the actual decorator function. The decorator function wraps the `my_function` by adding the pre-decoration logic to print the decorator argument.

When you run `my_function()` , it executes the wrapper function, which prints the decorator argument and then executes the original code inside `my_function` .

Example :

example of decorating functions with arguments is input validation in a web application. Let's consider a scenario where you have a web API endpoint that receives user input and needs to validate the input before processing it further.

Here's an example implementation:

```
def validate_input(*expected_args):
    def decorator(func):
        def wrapper(*args, **kwargs):
            # Perform input validation
            for arg, expected_type in zip(args, expected_args):
                if not isinstance(arg, expected_type):
                    raise ValueError(f"Invalid input type. Expected {expected_type.__name__}, but got {type(arg).__name__}.")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@validate_input(str, int)
def process_user_input(name, age):
    # Process the validated input
    print(f"User: {name}, Age: {age}")
```

```
# Test the decorated function
process_user_input("John Doe", 25)
```

In this example, the `validate_input` decorator factory takes the expected argument types as arguments. The decorator function, `decorator`, performs the input validation by checking the types of the arguments passed to the decorated function, `wrapper`. If any of the arguments do not match the expected types, a `ValueError` is raised.

The decorated function, `process_user_input`, is decorated with `@validate_input(str, int)`, indicating that the expected argument types are `str` (for the name) and `int` (for the age).

When you run this code and call `process_user_input("John Doe", 25)`, it will execute the wrapper function. The input values will be validated against the expected argument types. If the input is valid, the original function code inside `process_user_input` will be executed, printing the user's name and age. If the input does not match the expected types, a `ValueError` will be raised.

This example demonstrates how decorators with arguments can be used to validate user input in a web application, ensuring that the input adheres to the expected types or formats before further processing.

2.4 Chaining Multiple Decorators

You can apply multiple decorators to a function by stacking them using the decorator syntax. The decorators are applied from the bottom up, meaning the decorator closest to the function is applied first, followed by the next decorator, and so on. This allows you to apply multiple layers of behavior to the function.

Example:

```
@decorator1
@decorator2
@decorator3
def my_function():
    # Function code
```

Example:

```
import time

# Mock function for authentication check
def is_authenticated():
    return True

class UnauthorizedException(Exception):
    pass

class RateLimitExceededException(Exception):
    pass

def authenticate(func):
    def wrapper(*args, **kwargs):
        # Perform authentication logic
        if not is_authenticated():
            raise UnauthorizedException("Authentication failed")
        return func(*args, **kwargs)
    return wrapper

def rate_limit(max_requests, interval):
    def decorator(func):
        request_count = 0
        last_request_time = None

        def wrapper(*args, **kwargs):
            nonlocal request_count, last_request_time

            # Check rate limit
            if last_request_time and time.time() - last_request_time < interval:
                if request_count >= max_requests:
                    raise RateLimitExceededException("Rate limit exceeded")
            else:
                request_count = 0

            # Update request count and time
            request_count += 1
            last_request_time = time.time()

            return func(*args, **kwargs)
        return wrapper
    return decorator

@authenticate
```

```
@rate_limit(max_requests=100, interval=60)
def protected_api_endpoint():
    # API endpoint implementation
    return "Success"

# Test the chained decorators
response = protected_api_endpoint()
print(response)
```

In this example, the `authenticate` decorator is defined first, which performs the authentication logic. It checks if the user is authenticated before allowing access to the decorated function. If authentication fails, an exception is raised.

The `rate_limit` decorator is defined next, which enforces a rate limit on the decorated function. It limits the number of requests that can be made within a given interval. If the rate limit is exceeded, an exception is raised.

The `protected_api_endpoint` function is decorated with `@authenticate` and `@rate_limit(max_requests=100, interval=60)`, which means both decorators are applied to the function. When the function is called, it first goes through the authentication check, and if successful, the rate limit is enforced. If both checks pass, the function's implementation is executed and the result is returned.

When you run this code and call `protected_api_endpoint()`, it will execute the chained decorators. If the user is authenticated and the rate limit is not exceeded, the API endpoint implementation will be executed and the result will be printed.

This example demonstrates how chaining multiple decorators can be used in a real-world scenario to enforce both authentication and rate limiting for a web API endpoint. By applying multiple layers of decorators, you can add different levels of functionality or behavior to your functions, ensuring security, reliability, and performance in your application.

Please note that this is a simplified example meant to demonstrate the concept of chaining decorators in a real-world application. You would need to adapt the code and implement the actual authentication and rate limiting mechanisms specific to your application.

2.5 Handling Decorator Arguments and Options

Sometimes decorators themselves require options or additional arguments. In such cases, you can create a decorator that takes those arguments and returns the actual decorator function. This allows you to pass arguments to the decorator when applying it to a function.

Example:

```
def decorator_with_arguments(arg1, arg2):
    def decorator(func):
        def wrapper(*args, **kwargs):
            # Pre-decoration logic using arg1 and arg2
            print("Decorator arguments:", arg1, arg2)
            result = func(*args, **kwargs)
            # Post-decoration logic
            return result
        return wrapper
    return decorator

@decorator_with_arguments("value1", "value2")
def my_function():
    # Function code
```

2.6 Practical Use Cases and Examples

*** Authentication and authorization:**

Decorators can be used to enforce authentication and authorization checks on functions or endpoints.

Example

```
# Mock function for authentication check
def is_authenticated():
    return True

# Mock function for authorization check
def is_authorized_nurse():
    return True

# Mock function to get patient records
def get_patient_records(patient_id):
    return f"Patient records for ID: {patient_id}"
```

```
# Exception definition for unauthorized access
class UnauthorizedException(Exception):
    pass

# Decorator to enforce authentication
def authenticate(func):
    def wrapper(*args, **kwargs):
        if not is_authenticated():
            raise UnauthorizedException("Authentication failed")
        return func(*args, **kwargs)
    return wrapper

# Decorator to enforce nurse authorization
def authorize_nurse(func):
    def wrapper(*args, **kwargs):
        if not is_authorized_nurse():
            raise UnauthorizedException("You are not authorized to perform this action")
        return func(*args, **kwargs)
    return wrapper

# Decorated function to view patient records
@authenticate
@authorize_nurse
def view_patient_records(patient_id):
    # Functionality to view patient records
    return get_patient_records(patient_id)

# Test the decorated function
try:
    response = view_patient_records(patient_id="12345")
    print(response)
except UnauthorizedException as e:
    print(str(e))
```

In this example, the `authenticate` decorator ensures that the nurse is authenticated before accessing the `view_patient_records` function.
The `is_authenticated()` function can be implemented to perform the actual authentication check, such as verifying credentials or validating a token.

The `authorize_nurse` decorator checks if the authenticated nurse is authorized to perform the requested action. The `is_authorized_nurse()` function can be implemented to validate the nurse's role or permissions before granting access.

By applying both decorators to the `view_patient_records` function, the function will only be executed if the nurse is both authenticated and authorized. If any of the checks fail, an `UnauthorizedException` will be raised.

*** Input validation:**

Decorators can validate the input arguments passed to a function and raise an error if the validation fails.

Example

```
class InvalidInputException(Exception):
    pass

def validate_input(func):
    def wrapper(*args, **kwargs):
        # Perform input validation
        validate_patient_id(kwargs.get("patient_id"))
        validate_vital_signs(kwargs.get("vital_signs"))
        return func(*args, **kwargs)
    return wrapper

def validate_patient_id(patient_id):
    # Check if patient ID is valid
    if not patient_id:
        raise InvalidInputException("Invalid patient ID")

def validate_vital_signs(vital_signs):
    # Check if vital signs are valid
    if not vital_signs or not isinstance(vital_signs, dict):
        raise InvalidInputException("Invalid vital signs")

@validate_input
def record_vital_signs(patient_id, vital_signs):
    # Functionality to record vital signs for a patient
    print(f"Recording vital signs for Patient ID: {patient_id}")
    print("Vital signs:", vital_signs)

# Test the decorated function with valid input
record_vital_signs(patient_id="12345", vital_signs={"temperature": 98.6, "heart_rate": 75})

# Test the decorated function with invalid input
record_vital_signs(patient_id="", vital_signs=None)
```

In this example, the `validate_input` decorator is defined to validate the input arguments passed to the `record_vital_signs` function. Inside the decorator, two input validation functions (`validate_patient_id` and `validate_vital_signs`) are called to validate the `patient_id` and `vital_signs` arguments, respectively.

The `validate_patient_id` function checks if the patient ID is valid, and the `validate_vital_signs` function checks if the vital signs argument is valid (in this case, a dictionary containing vital sign measurements).

The `record_vital_signs` function is decorated with `@validate_input` , which applies the input validation checks before executing the function's functionality.

When you run this code, it will execute the decorated function with both valid and invalid inputs. If the input passes the validation checks, the function's implementation will be executed and the vital signs will be recorded. If any of the input validations fail, an `InvalidInputException` will be raised, indicating the specific input error.

This example demonstrates how decorators can be used to perform input validation in an EMR application for nurses. By applying the `validate_input` decorator to relevant functions, you can ensure that the input arguments meet the required criteria before further processing or recording of patient data.

*** Caching:**

Decorators can cache the results of expensive function calls, improving performance by avoiding redundant computations.

Example:

```
import functools

# Cache dictionary to store the results
cache = {}

def memoize(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # Generate a unique cache key based on function arguments
        cache_key = (func.__name__, args, frozenset(kwargs.items()))

        # Check if the result is already in the cache
        if cache_key in cache:
            print("Result retrieved from cache")
            return cache[cache_key]

        # Execute the function and store the result in the cache
        result = func(*args, **kwargs)
        cache[cache_key] = result
        return result

    return wrapper

@memoize
def expensive_computation(num):
    # Simulate an expensive computation
    print("Performing expensive computation...")
    return num * 2

# Test the decorated function
print(expensive_computation(5))
print(expensive_computation(5))
print(expensive_computation(10))
print(expensive_computation(10))
```

In this example, the `memoize` decorator is defined to cache the results of the `expensive_computation` function. The cache is represented by a dictionary named `cache` , which stores the results using a unique cache key.

The `wrapper` function checks if the result is already present in the cache. If it is, the result is retrieved from the cache and returned immediately. Otherwise, the function is executed, and the result is stored in the cache with the corresponding cache key.

The `functools.wraps` decorator is used to preserve the metadata (such as the function name) of the original function in the wrapper function.

When you run this code, you'll notice that the first time the `expensive_computation` function is called with a specific argument, it performs the expensive computation and stores the result in the cache. The second time the same argument is passed, the result is retrieved directly from the cache, avoiding the redundant computation.

This caching technique can significantly improve performance when dealing with expensive computations or function calls that are frequently executed with the same set of arguments. By caching the results, subsequent calls with the same arguments can be served from the cache, reducing the computational overhead.

Please note that in a real-world application, you may need to consider cache eviction strategies, such as setting a maximum cache size or implementing cache expiry based on time or other criteria, depending on the specific requirements of your application.

*** Logging:**

Decorators can log function calls, providing information about when and how the function was invoked.

```
import logging
import functools

# Configure the logging module
logging.basicConfig(level=logging.INFO)

def log_function_call(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # Log the function call
        logging.info(f"Function {func.__name__} called with args: {args}, kwargs: {kwargs}")

        # Execute the function
        result = func(*args, **kwargs)

        # Log the function result
        logging.info(f"Function {func.__name__} returned: {result}")

        return result

    return wrapper

@log_function_call
def calculate_sum(a, b):
    return a + b

# Test the decorated function
calculate_sum(5, 10)
```

In this example, the `log_function_call` decorator is defined to log the function calls and their results. The logging module is configured to log messages at the `INFO` level.

The `wrapper` function logs the function call before executing the function and logs the function result after the execution. The logging messages include the function name, arguments passed (`args` and `kwargs`), and the returned result.

The `functools.wraps` decorator is used to preserve the metadata of the original function in the wrapper function.

When you run this code, you'll see log messages indicating when and how the `calculate_sum` function was called, along with the returned result. This logging information can be useful for debugging, tracking function invocations, or monitoring the behavior of your application.

By applying the `log_function_call` decorator to relevant functions, you can easily add logging functionality to them without modifying their implementation directly. This allows you to gain insights into the function's usage, input arguments, and output results, which can be valuable in understanding the flow and behavior of your application.

Please note that in a real-world application, you may want to customize the logging format, level, and destination based on your specific requirements. The example provided gives a basic understanding of how decorators can be used for logging function calls.

*** Timing and profiling:**

Decorators can measure the execution time of functions to identify performance bottlenecks.

Example:

```
import time
import functools

def profile(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()

        # Execute the function
        result = func(*args, **kwargs)

        end_time = time.time()
        execution_time = end_time - start_time

        print(f"Function {func.__name__} took {execution_time:.4f} seconds to execute.")

        return result

    return wrapper

@profile
```

```
def calculate_sum(numbers):
    total = sum(numbers)
    return total

# Test the decorated function
calculate_sum([1, 2, 3, 4, 5])
```

In this example, the `profile` decorator is defined to measure the execution time of a function. The `time` module is used to track the start and end times of function execution.

The `wrapper` function records the start time before executing the function and calculates the execution time by subtracting the start time from the end time. The execution time is then printed to the console.

The `functools.wraps` decorator is used to preserve the metadata of the original function in the wrapper function.

When you run this code, you'll see the execution time printed to the console after the `calculate_sum` function is called. This information can be useful for identifying bottlenecks or optimizing performance-critical sections of your code.

Example

```
import time

def log(message):
    # Placeholder function for logging
    print(message)

def profile_performance(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        log(f"Function: {func.__name__}, Execution Time: {execution_time} seconds")
        return result
    return wrapper

@profile_performance
def search_patients(query):
    # Functionality to search for patients based on a query
    # Placeholder implementation with fake data
    patients = [
        {"id": 1, "name": "John Doe", "age": 30},
        {"id": 2, "name": "Jane Smith", "age": 45},
        {"id": 3, "name": "Michael Johnson", "age": 55},
        {"id": 4, "name": "Emily Davis", "age": 28},
        {"id": 5, "name": "William Brown", "age": 60}
    ]

    # Simulating search operation
    filtered_patients = [patient for patient in patients if query.lower() in patient["name"].lower()]

    return filtered_patients

# Test the decorated function
results = search_patients("john")
print(results)
```

In this example, the `profile_performance` decorator is defined to measure the execution time of the `search_patients` function. The `wrapper` function records the start time, executes the function, calculates the execution time, and logs the result.

The `search_patients` function is implemented with fake patient data and performs a search operation based on a given query. The filtered patients are returned as a result.

When you run this code, you'll see the execution time printed to the console after the `search_patients` function is called, along with the filtered patient results.

Please note that the `log` function used in the `profile_performance` decorator is a placeholder for logging. You can replace it with an actual logging mechanism suitable for your application, such as using the `logging` module or writing logs to a file or database.

This example demonstrates how the `profile_performance` decorator can be used to measure the execution time of the `search_patients` function

By applying the `profile` decorator to relevant functions, you can easily measure and monitor their execution time without modifying their implementation directly. This allows you to gain insights into the performance characteristics of your code and identify areas that may require optimization.

Please note that in a real-world application, you may want to use more advanced profiling tools and techniques to get detailed performance analysis. The example provided gives a basic understanding of how decorators can be used for timing and profiling, but for more advanced profiling, consider using specialized profiling libraries or tools available in your programming language.

*** Retry logic:**

Decorators can implement retry mechanisms for functions that may fail temporarily.

Example:

```
import time
import functools

def retry(max_attempts=3, delay=1):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            attempts = 0

            while attempts < max_attempts:
                try:
                    result = func(*args, **kwargs)
                    return result
                except Exception as e:
                    print(f"Attempt {attempts+1} failed. Error: {str(e)}")
                    attempts += 1
                    time.sleep(delay)

            raise Exception(f"Function {func.__name__} failed after {max_attempts} attempts.")

        return wrapper

    return decorator

@retry(max_attempts=5, delay=2)
def connect_to_server():
    # Simulating a connection to a server
    # Placeholder implementation with a random error
    import random
    if random.random() < 0.5:
        raise Exception("Connection error")
    else:
        print("Connected to the server")

# Test the decorated function
connect_to_server()
```

In this example, the `retry` decorator is defined to implement a retry mechanism for the `connect_to_server` function. The decorator accepts two parameters: `max_attempts` (the maximum number of retry attempts) and `delay` (the delay between each retry attempt).

The `wrapper` function encapsulates the retry logic. It executes the function within a while loop and catches any exceptions raised. If the function executes successfully, the result is returned. If an exception occurs, the number of attempts is incremented, and a delay is introduced before the next attempt. The process continues until either the function succeeds or the maximum number of attempts is reached.

When you run this code, the `connect_to_server` function is decorated with the `retry` decorator, which allows it to be retried multiple times in case of a temporary connection error. The number of attempts and any error messages are printed to the console.

Please note that this example uses a random error to simulate a connection failure. In a real-world application, you would replace the placeholder implementation with the actual code that may encounter temporary failures, such as network connections, API calls, or database connections.

The retry logic provided by the decorator can be useful in scenarios where temporary failures are expected, and retrying the operation may result in a successful outcome. It helps to handle transient issues and improve the robustness of your code by automatically retrying failed operations.

You can customize the maximum number of attempts and the delay between retries based on your specific requirements. Additionally, you can handle different types of exceptions separately, introduce exponential backoff strategies, or implement more sophisticated retry mechanisms depending on the needs of your application.

*** Decorators in web frameworks:**

Web frameworks like Flask and Django use decorators extensively for URL routing, view authorization, and request/response handling.

```
from flask import Flask, request, redirect
from functools import wraps

app = Flask(__name__)
```

```
# Placeholder implementation for is_user_authenticated() function
def is_user_authenticated():
    # Your authentication logic goes here
    # Return True if the user is authenticated, False otherwise
    # Replace this implementation with your actual authentication logic
    return False

def login_required(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if not is_user_authenticated():
            return redirect('/login')
        return func(*args, **kwargs)
    return wrapper

@app.route('/dashboard')
@login_required
def dashboard():
    # Functionality to render the dashboard
    return "Welcome to the dashboard!"

@app.route('/profile')
@login_required
def profile():
    # Functionality to render the user profile
    return "User profile page"

if __name__ == '__main__':
    app.run()
```

In this example, the Flask web framework is used, and two routes (`/dashboard` and `/profile`) are defined. The `login_required` decorator is used to enforce authentication before accessing these routes.

The `login_required` decorator checks if the user is authenticated by calling the `is_user_authenticated()` function. If the user is not authenticated, it redirects them to the `/login` page. Otherwise, it allows the execution of the decorated function.

The `@app.route` decorator is provided by Flask itself and is used to map a URL path to a specific view function. In this case, the `dashboard` and `profile` functions are decorated with `@app.route` to associate them with their respective URL paths.

By applying the `@login_required` decorator to the `dashboard` and `profile` functions, the authentication check is automatically performed before these routes are accessed. This ensures that only authenticated users can access these protected views.

This is just a simplified example, but in real-world web frameworks, decorators are used extensively for various purposes, including URL routing, view authorization, request/response handling, caching, error handling, and much more. Decorators provide a clean and flexible way to extend and modify the behavior of web framework components without modifying their core implementation.

By leveraging decorators, web frameworks enable developers to build robust and secure web applications efficiently, handling authentication, authorization, and other aspects of web development seamlessly.

These practical examples demonstrate the versatility and power of function decorators in Python. By leveraging decorators, you can enhance the functionality, behavior, and reliability of your functions in various real-world scenarios.

2.7 decorator factories creating decorators with configuration

Decorator factories are functions that generate decorator instances with configurable behavior. They allow you to create decorators that can be customized with specific configurations or options.

To create a decorator factory, you define a function that takes the desired configuration parameters as arguments and returns a decorator function. The decorator function, in turn, takes the target function or class as its argument and returns a modified version of it.

Here's an example of creating a decorator factory:

```
def decorator_factory(config_param):
    def decorator(target):
        # Modify the target based on the configuration parameter
        def wrapper(*args, **kwargs):
            # Perform decoration logic
            result = target(*args, **kwargs)
            # Additional logic based on the configuration
            print("Config parameter:", config_param)
            return result
        return wrapper
    return decorator

# Create a decorator with specific configuration
```

```
my_decorator = decorator_factory("example_config")

# Apply the decorator to a function or class
@my_decorator
def my_function():
    print("Function called")

# Test the decorated function
my_function()  # Output: "Function called" followed by "Config parameter: example_config"
```

In this example, the `decorator_factory` function takes a configuration parameter `config_param` and returns a decorator function. The decorator function, `decorator`, takes the target function or class as its argument and returns a modified version (`wrapper`) with additional logic based on the configuration parameter.

Example:

```
import logging

def logging_decorator_factory(log_level):
    def logging_decorator(func):
        logger = logging.getLogger(func.__name__)
        logger.setLevel(log_level)

        def wrapper(*args, **kwargs):
            logger.info(f"Executing {func.__name__}...")
            result = func(*args, **kwargs)
            logger.info(f"{func.__name__} executed.")
            return result

        return wrapper

    return logging_decorator

# Create a logging decorator with different log levels
debug_logging_decorator = logging_decorator_factory(logging.DEBUG)
info_logging_decorator = logging_decorator_factory(logging.INFO)

# Apply the logging decorators to functions
@debug_logging_decorator
def calculate_sum(a, b):
    return a + b

@info_logging_decorator
def calculate_product(a, b):
    return a * b

# Test the decorated functions
print(calculate_sum(2, 3))  # Output: 5
print(calculate_product(2, 3))  # Output: 6
```

In this example, the `logging_decorator_factory` is a decorator factory that takes a log level as its argument. It returns a decorator function, `logging_decorator`, which is responsible for logging the execution of the target function. The log level is set based on the configuration provided.

By using the decorator factory, you can create different logging decorators with different log levels. In the example, we create a `debug_logging_decorator` with `DEBUG` log level and an `info_logging_decorator` with `INFO` log level.

When the decorated functions (`calculate_sum` and `calculate_product`) are called, the logging decorators log the execution information based on the configured log level. This allows you to have fine-grained control over the logging behavior of each function independently.

By using a decorator factory, you can generate decorators with different configurations and apply them to different functions or classes as needed. This allows you to create reusable and customizable decorators that can adapt to different scenarios or requirements.

2.8 dynamic decorators generating decorators at runtime

Dynamic decorators refer to the generation of decorators at runtime, where the behavior of the decorator is determined dynamically based on certain conditions or inputs. This approach allows you to create decorators with varying functionality based on runtime information.

Here's an example of dynamically generating decorators at runtime:

```
def dynamic_decorator_generator(condition):
    def dynamic_decorator(func):
        if condition:
            # Decorate the function in a specific way
            def wrapper(*args, **kwargs):
                # Perform decoration logic based on the condition
                print("Condition is True")
```

```
                return func(*args, **kwargs)
            return wrapper
        else:
            # Decorate the function differently
            def wrapper(*args, **kwargs):
                # Perform alternative decoration logic
                print("Condition is False")
                return func(*args, **kwargs)
            return wrapper
    return dynamic_decorator

# Generate a decorator based on a condition
condition = True  # Example condition
decorator = dynamic_decorator_generator(condition)

# Apply the generated decorator to a function
@decorator
def my_function():
    print("Function called")

# Test the decorated function
my_function()  # Output: "Condition is True" followed by "Function called"
```

In this example, the `dynamic_decorator_generator` is a function that takes a condition as an input. Based on the condition, it dynamically generates and returns a decorator function (`dynamic_decorator`) with different functionality.

When the decorator is applied to the `my_function` , the decorator function is chosen at runtime based on the condition. If the condition is `True` , the `wrapper` function within the decorator performs one set of decoration logic. If the condition is `False` , an alternative set of decoration logic is applied.

Example:

A real-world application of dynamically generating decorators at runtime can be seen in feature toggling or feature flagging in software development.

Feature toggling is a technique used to enable or disable certain features or functionality in an application based on runtime conditions or configuration. It allows developers to control the availability of features without modifying the codebase.

Here's an example of how dynamic decorators can be used for feature toggling:

```
def feature_toggle(feature_name):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if is_feature_enabled(feature_name):
                return func(*args, **kwargs)
            else:
                raise FeatureNotEnabledException(f"Feature '{feature_name}' is not enabled.")
        return wrapper
    return decorator

# Feature toggle configuration
feature_flags = {
    'analytics': True,
    'payment': False,
}

# Check if a feature is enabled
def is_feature_enabled(feature_name):
    return feature_flags.get(feature_name, False)

# Custom exception for disabled features
class FeatureNotEnabledException(Exception):
    pass

# Define a feature-dependent function
@feature_toggle('analytics')
def track_user_activity(user_id):
    print(f"Tracking user activity for user ID: {user_id}")

# Test the decorated function
track_user_activity(user_id='123')  # Output: "Tracking user activity for user ID: 123"

# Define another feature-dependent function
@feature_toggle('payment')
def process_payment(amount):
    print(f"Processing payment for amount: {amount}")

# Test the decorated function
process_payment(amount=100)  # Raises FeatureNotEnabledException: "Feature 'payment' is not enabled."
```

In this example, the `feature_toggle` decorator factory generates decorators based on the feature name. The decorator checks if the specified feature is enabled or not by looking up its state in the `feature_flags` configuration.

When a function is decorated with a feature toggle decorator, the decorator's `wrapper` function is executed. If the feature is enabled, the decorated function is called and its functionality is executed. If the feature is disabled, a `FeatureNotEnabledException` is raised to indicate that the feature is not available.

This approach allows you to control the availability of features in your application dynamically. You can toggle the features on or off by changing the feature flags in the configuration, without modifying the code of the decorated functions. It provides a way to enable or disable features in production, perform A/B testing, or gradually roll out new features to specific users or groups.

By dynamically generating decorators at runtime, you can create flexible and adaptive decoration behavior based on runtime conditions, inputs, or configurations. This approach allows you to tailor the behavior of decorators to specific runtime scenarios.

2.9 decorator stacking and ordering

Decorator stacking refers to the practice of applying multiple decorators to a function or class in a specific order. Each decorator adds additional behavior or modifications to the target function or class, resulting in a chain of decorators that modify the behavior or properties of the target.

The order in which decorators are applied can be significant because each decorator may depend on the modifications made by the previous decorators in the stack. Therefore, understanding the order of execution of decorators is crucial to ensure the desired behavior and functionality.

Here's an example that demonstrates decorator stacking and ordering:

```
def decorator1(func):
    def wrapper(*args, **kwargs):
        print("Decorator 1 - Before function execution")
        result = func(*args, **kwargs)
        print("Decorator 1 - After function execution")
        return result
    return wrapper

def decorator2(func):
    def wrapper(*args, **kwargs):
        print("Decorator 2 - Before function execution")
        result = func(*args, **kwargs)
        print("Decorator 2 - After function execution")
        return result
    return wrapper

@decorator1
@decorator2
def my_function():
    print("Function executed")

# Test the decorated function
my_function()
```

In this example, we have two decorators: `decorator1` and `decorator2`. The `@decorator1` is applied first, followed by `@decorator2` on the `my_function`. This order is determined by the order of the decorators directly above the function declaration.

When `my_function` is called, the decorators are executed in the reverse order of their application. In this case, `decorator2` is executed first, followed by `decorator1`. This order allows `decorator2` to wrap `decorator1`, which then wraps the original `my_function`.

The output of running this code will be:

```
Decorator 1 - Before function execution
Decorator 2 - Before function execution
Function executed
Decorator 2 - After function execution
Decorator 1 - After function execution
```

As you can see, the decorators are stacked in reverse order, with the innermost decorator (`decorator2`) being executed first and the outermost decorator (`decorator1`) being executed last.

Example

```
from flask import Flask, request, redirect
from functools import wraps

app = Flask(__name__)
```

```
def is_user_authenticated():
    # Placeholder function to check if the user is authenticated
    # Replace with your own authentication logic
    return True

def is_user_admin():
    # Placeholder function to check if the user is an admin
    # Replace with your own admin authorization logic
    return False

def login_required(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if not is_user_authenticated():
            return redirect('/login')
        return func(*args, **kwargs)
    return wrapper

def admin_required(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if not is_user_admin():
            return redirect('/dashboard')
        return func(*args, **kwargs)
    return wrapper

@app.route('/dashboard')
@login_required
def dashboard():
    # Functionality to render the dashboard
    return "Welcome to the dashboard!"

@app.route('/admin')
@login_required
@admin_required
def admin_panel():
    # Functionality to render the admin panel
    return "Admin panel"

if __name__ == '__main__':
    app.run()
```

In this example, we have two decorators: `login_required` and `admin_required`. The `@login_required` decorator is applied first, followed by `@admin_required` on the `admin_panel` route. This order ensures that the authentication check is performed before the admin authorization check.

When a user accesses the `/admin` route, the decorators are executed in the order they are stacked. First, the `login_required` decorator checks if the user is authenticated. If the user is not authenticated, they are redirected to the login page. If the user is authenticated, the execution proceeds to the `admin_required` decorator, which checks if the user is an admin. If the user is not an admin, they are redirected to the dashboard.

By stacking and ordering the decorators in this way, we can enforce authentication and authorization checks on specific routes. The decorators allow us to modularly add and combine these checks to handle various access control scenarios in our web application.

This real-life example demonstrates how decorator stacking and ordering can be used in web frameworks to handle route authorization. It ensures that the necessary checks are performed in the correct order, providing the desired functionality and access control for different parts of the application.

By understanding the order of execution, you can control the flow of behavior and modifications applied by each decorator in the stack. This allows you to build complex functionality by combining multiple decorators in a specific order to achieve the desired outcome.

2.10 Resolving conflicts in decorator chains

Resolving conflicts in decorator chains can be achieved by properly ordering the decorators and ensuring that they are applied in the desired sequence. Here are a few approaches to resolve conflicts in decorator chains:

1. **Order the decorators correctly**: Decorators are applied in the order they are written, from top to bottom. If you have multiple decorators that might conflict with each other, make sure to arrange them in the desired order so that the outermost decorator is applied first and the innermost decorator is applied last. This order ensures that the decorators wrap the function or class in the intended sequence.
2. **Use class-level decorators**: If you are working with class decorators, you can define class-level decorators to ensure that they are applied consistently to all methods within the class. This way, you don't have to worry about conflicting decorators for individual methods.
3. **Modify decorator implementations:** If conflicts arise due to incompatible behaviors between decorators, you may need to modify the implementation of the decorators to ensure they work harmoniously together. This might involve adjusting the logic within the decorators or creating a new decorator that combines the desired behavior of multiple decorators.
4. **Use decorator factories**: Decorator factories allow you to generate decorators dynamically with specific configurations. By customizing the parameters or behavior of the decorator factories, you can create decorators that handle conflicts or specific requirements more effectively.

Example:

```
from functools import wraps

def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Decorator 1 logic
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Decorator 2 logic
        return func(*args, **kwargs)
    return wrapper

@decorator1
@decorator2
def my_function():
    # Function logic
    pass
```

In this example, the `@wraps` decorator is used to wrap the inner wrapper functions within each decorator. This ensures that the original function name and attributes are preserved.

Real-world example:

Let's consider a real-world example where decorators are used to handle request validation and logging in a web application.

```
from functools import wraps
from flask import Flask, request, jsonify

app = Flask(__name__)

def validate_request_data(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Request data validation logic
        data = request.get_json()
        if not data:
            return jsonify({'message': 'Invalid request data'}), 400
        return func(*args, **kwargs)
    return wrapper

def log_request(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Logging logic
        print(f"Received request: {request.method} {request.url}")
        return func(*args, **kwargs)
    return wrapper

@app.route('/api/endpoint', methods=['POST'])
@validate_request_data
@log_request
def api_endpoint():
    # Endpoint logic
    data = request.get_json()
    return jsonify({'message': f"Received data: {data}"}), 200

if __name__ == '__main__':
    app.run()
```

In this example, the `validate_request_data` decorator validates the request data before executing the endpoint logic. The `log_request` decorator logs the incoming request details. By using the `@wraps` decorator, conflicts between the decorators are resolved, and the original function's attributes are preserved.

This pattern can be helpful in scenarios where you want to apply multiple decorators to a function in a specific order, while ensuring that the function's original attributes are maintained.

Remember that the specific approach to resolving conflicts in decorator chains depends on the nature of the conflict and the behavior you want to achieve. Careful consideration of the order of decorators and their interactions will help ensure that the desired functionality is applied correctly.

2.11 Conditional and parameterized decorator composition

Conditional and parameterized decorator composition allows you to dynamically apply decorators based on certain conditions or pass parameters to decorators for customization. This flexibility allows you to apply different sets of decorators based on runtime conditions or configure decorators with specific settings.

Here's an example that demonstrates conditional and parameterized decorator composition:

```
from functools import wraps

def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Decorator 1 logic
        return func(*args, **kwargs)
    return wrapper

def decorator2(parameter):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Decorator 2 logic with parameter
            print(f"Decorator 2 parameter: {parameter}")
            return func(*args, **kwargs)
        return wrapper
    return decorator

def decorator_factory(condition):
    if condition:
        return decorator1
    else:
        return decorator2("Custom Parameter")

@decorator_factory(True)
def my_function():
    # Function logic
    print("Inside my_function")

my_function()  # Add this line to invoke my_function and see the output
```

In this example, the `decorator_factory` function returns different decorators based on the condition. If the condition is `True`, it returns `decorator1`, and if the condition is `False`, it returns `decorator2` with a custom parameter.

2.12 Creating decorators DLS (domain Specific Languages)

Domain-Specific Languages (DSLs) are specialized programming languages designed for specific domains or problem areas. In the context of decorators, creating a DSL involves designing decorators with a specific syntax and behavior that aligns with the requirements of a particular domain.

Here's an example of creating a DSL for logging decorators:

```
from functools import wraps

def log_action(action):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(f"Performing {action}...")
            result = func(*args, **kwargs)
            print(f"{action} completed.")
            return result
        return wrapper
    return decorator

@log_action("database query")
def perform_database_query():
    print("Executing database query...")
    # Simulating some database query logic
    data = ["result 1", "result 2", "result 3"]
    return data

@log_action("file processing")
def process_file(file_path):
    print(f"Processing file: {file_path}...")
    # Simulating some file processing logic
    return True

# Calling the decorated functions
result = perform_database_query()
print("Query results:", result)

file_processed = process_file("example.txt")
print("File processed:", file_processed)
```

In the above example, the `log_action` decorator factory is designed as a DSL for creating logging decorators. It takes an `action` parameter, which represents the action being performed, and returns a decorator that logs the start and completion of the action.

Example

```
class Router:
    routes = {}

    @classmethod
    def route(cls, url, methods=['GET']):
        def decorator(func):
            cls.routes[url] = {
                'func': func,
                'methods': methods
            }
            return func
        return decorator

class App:
    def __init__(self):
        self.router = Router()

    def add_route(self, url, methods=['GET']):
        def decorator(func):
            self.router.routes[url] = {
                'func': func,
                'methods': methods
            }
            return func
        return decorator

app = App()

# Using the DSL provided by the Router class
@Router.route('/home', methods=['GET'])
def home_page():
    return 'Welcome to the home page'

# Using the DSL provided by the App instance
@app.add_route('/about', methods=['GET', 'POST'])
def about_page():
    return 'This is the about page'

if __name__ == '__main__':
    # Access the routes defined in the router
    print(Router.routes)

    # Access the routes defined in the app instance
    print(app.router.routes)
```

In this example, we define a `Router` class with a `route` class method that acts as a decorator factory. This method takes a URL and HTTP methods as arguments and returns a decorator function. When the decorator is applied to a function, it stores the function and its associated URL and methods in the `routes` dictionary of the `Router` class.

We also create an `App` class that has an `add_route` instance method, which works similarly to the `Router.route` method. The difference is that it stores the routes in the `routes` dictionary of the `App` instance.

By using the DSL provided by either the `Router` class or the `App` instance, we can easily define routes in our web application using decorators. The routes are stored in the respective `routes` dictionaries, which can be accessed and used to handle incoming requests.

By using the DSL, you can easily create logging decorators for specific actions by providing the action name as a parameter when applying the decorator. The resulting decorators will have the desired behavior and logging functionality specific to the action.

Using DSLs in decorators allows you to create a more expressive and readable syntax that aligns with the requirements of your specific domain. It provides a way to encapsulate common functionality and easily apply it to functions or methods based on their purpose or context.

2.13 Handling Complex Decorator Nesting

Handling complex decorator nesting involves applying multiple decorators to a function or class in a specific order to achieve the desired behavior. *Here's an example that demonstrates handling complex decorator nesting in Python:*

```
def decorator1(func):
    def wrapper(*args, **kwargs):
        print("Decorator 1")
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    def wrapper(*args, **kwargs):
        print("Decorator 2")
        return func(*args, **kwargs)
    return wrapper

def decorator3(func):
    def wrapper(*args, **kwargs):
        print("Decorator 3")
```

```
        return func(*args, **kwargs)
    return wrapper

@decorator1
@decorator2
@decorator3
def my_function():
    print("Function logic")

my_function()
```

In this example, we have three decorators: `decorator1`, `decorator2`, and `decorator3`. The `my_function` is decorated with these decorators in a specific order. When `my_function` is called, the decorators are applied in reverse order, starting from the top-most decorator.

The expected output of the above code will be:

```
Decorator 1
Decorator 2
Decorator 3
Function logic
```

The decorators are executed in the order of `decorator3`, `decorator2`, and finally `decorator1`. This order allows the decorators to wrap around the function, with each decorator adding its own behavior or functionality.

Example:

```
from functools import wraps
from flask import Flask, request, jsonify

app = Flask(__name__)

# Example implementation of user authentication logic
def is_user_authenticated():
    # Check if the user is authenticated
    # Return True or False based on the authentication status
    # Replace this with your actual implementation
    return True

# Example implementation of user authorization logic
def is_user_authorized(roles):
    # Check if the user has the required roles/permissions
    # Return True or False based on the authorization status
    # Replace this with your actual implementation
    return True

def authenticate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Perform authentication logic
        if not is_user_authenticated():
            return jsonify({'message': 'Authentication failed'}), 401
        return func(*args, **kwargs)
    return wrapper

def authorize(roles):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Perform authorization logic
            if not is_user_authorized(roles):
                return jsonify({'message': 'Unauthorized'}), 403
            return func(*args, **kwargs)
        return wrapper
    return decorator

@app.route('/dashboard')
@authenticate
@authorize(['admin'])
def dashboard():
    # Functionality to render the dashboard
    return jsonify({'message': 'Welcome to the dashboard'})

@app.route('/profile')
@authenticate
def profile():
    # Functionality to render the user profile
    return jsonify({'message': 'User profile page'})

if __name__ == '__main__':
    app.run()
```

In this example, the `authenticate` decorator ensures that only authenticated users can access the routes it is applied to. The `authorize` decorator, which takes a list of roles as a parameter, checks if the authenticated user has the required roles to access the routes. By nesting these decorators, you can apply both authentication and authorization checks to specific routes, providing a secure and controlled access mechanism.

This is just one example of how decorators can be used in a real-world application. Depending on the requirements, decorators can be nested and combined to implement a wide range of functionalities, making the code modular, reusable, and easier to maintain.

By nesting decorators in a specific order, you can build complex behavior and achieve the desired functionality in your code. This approach is commonly used in web frameworks, middleware implementations, and other scenarios where multiple layers of functionality need to be applied to functions or classes.

3 Class Decorators

3.1 Decorating classes with decorators

Decorating classes with decorators: Similar to function decorators, class decorators are used to modify or enhance the behavior of classes. They are applied to the class definition itself, rather than individual methods or attributes within the class. Class decorators are typically defined as functions that accept a class as an argument and return a modified or enhanced version of that class.

Here's an example of decorating a class with a class decorator:

```
def add_custom_method(cls):
    cls.custom_method = lambda self: print("Custom method called")
    return cls

@add_custom_method
class MyClass:
    pass

obj = MyClass()
obj.custom_method()  # Output: "Custom method called"
```

In this example, the `add_custom_method` class decorator adds a `custom_method` to the `MyClass` class. By applying the `@add_custom_method` decorator to the class definition, the class is modified to include the custom method. Instances of the class can then call this method.

3.2 — Class decorators vs. function decorators

The main difference between class decorators and function decorators lies in what they can modify. **Function decorators** are applied to individual functions or methods, allowing you to modify their behavior or add additional functionality. *On the other hand,* **class decorators** operate on the class level, modifying the class itself or adding new attributes or methods to the class.

3.3 — Use cases for class decorators

Class decorators can be used in various scenarios to modify or enhance the behavior of classes. Here are some common use cases for class decorators:

1. **Adding or modifying class attributes**: Class decorators can be used to dynamically add or modify class attributes. For example, you can create a decorator that automatically adds certain attributes or properties to a class based on certain conditions or configurations.

2. **Implementing mixins**: Mixins are reusable components that provide specific functionality to classes. Class decorators can be used to apply mixins to classes, allowing them to inherit the desired behavior without explicitly subclassing or duplicating code.

3. **Registering classes in a registry**: Decorators can be used to automatically register classes in a registry or container. This is particularly useful in cases where you want to maintain a collection of classes for later use, such as plugins or extensions.

4. **Enforcing class contracts or interfaces:** Decorators can be used to enforce certain contracts or interfaces on classes. They can perform runtime checks to ensure that a class adheres to a specific interface or meets certain requirements, providing validation or enforcing certain behaviors.

5. **Caching class instances:** Decorators can implement caching mechanisms for class instances, allowing you to reuse previously instantiated objects instead of creating new ones. This can be useful in scenarios where object creation is expensive or when you want to limit the number of instances.

6. **Profiling or logging class behavior**: Decorators can be used to add profiling or logging functionality to classes, allowing you to monitor and analyze the behavior of class methods or attributes. This can help with debugging, performance optimization, or gathering usage statistics.

These are just a few examples of the many use cases for class decorators. The flexibility and power of decorators make them a versatile tool for modifying and extending class behavior in Python.

Example:

```
from flask import Flask

app = Flask(__name__)

class Route:
    def __init__(self, url, methods):
        self.url = url
        self.methods = methods

    def __call__(self, view_func):
        app.add_url_rule(self.url, view_func=view_func, methods=self.methods)
        return view_func

@Route('/home', methods=['GET'])
def home_page():
    return 'Welcome to the home page'

@Route('/about', methods=['GET', 'POST'])
def about_page():
    if request.method == 'GET':
        return 'About page (GET)'
    elif request.method == 'POST':
        return 'About page (POST)'

if __name__ == '__main__':
    app.run()
```

In this example, the `Route` class is a decorator class that accepts the URL and HTTP methods as parameters. When the decorator is applied to a view function, it registers the URL and associates the view function with the specified HTTP methods using `app.add_url_rule()`.

By using the `Route` decorator class, we can easily define multiple routes and their corresponding view functions in a clean and readable way. The class decorator allows us to encapsulate the routing logic and reuse it across different view functions.

This is just one example of how class decorators can be used in a real-world application. Class decorators provide a way to extend and customize the behavior of classes, making them a powerful tool for building flexible and modular systems.

3.4 Enhancing Class Behavior with Decorators

Example

One real-world application of enhancing class behavior with decorators is in the field of data validation and input sanitization. Let's consider a scenario where we have a class representing a user registration form, and we want to ensure that the input data is valid and sanitized before processing it.

Here's an example:

```
def validate_input(func):
    def wrapper(self, *args, **kwargs):
        # Perform validation logic on input data
        if not self.is_valid():
            raise ValueError('Invalid input data')
        return func(self, *args, **kwargs)
    return wrapper

def sanitize_input(func):
    def wrapper(self, *args, **kwargs):
        # Perform sanitization logic on input data
        self.sanitize_data()
        return func(self, *args, **kwargs)
    return wrapper

class UserRegistrationForm:
    def __init__(self, username, password, email):
        self.username = username
        self.password = password
        self.email = email

    @validate_input
    @sanitize_input
    def process_registration(self):
        # Process the registration logic
        print('User registration processed successfully')

    def is_valid(self):
        # Validate the input data
        return all([self.username, self.password, self.email])

    def sanitize_data(self):
        # Sanitize the input data
        self.username = self.username.strip()
        self.email = self.email.lower()

# Usage
```

```
form = UserRegistrationForm('  john_doe  ', 'password123', 'john@example.com')
form.process_registration()
```

In this example, we have two decorators: `validate_input` and `sanitize_input`. The `validate_input` decorator validates the input data by checking if all the required fields are provided, and the `sanitize_input` decorator sanitizes the input data by removing leading/trailing whitespace and converting the email to lowercase.

By applying these decorators to the `process_registration` method, we ensure that the input data is validated and sanitized before further processing. This helps in maintaining data integrity and improving the overall reliability of the registration process.

This is just one example of how decorators can enhance class behavior in a real-world application. Decorators provide a way to add additional functionality to methods, such as input validation, caching, logging, and more, without modifying the original class code. They help in keeping the code modular, reusable, and easier to maintain.

3.5 Implementing Decorator Stacks and Order of Execution

Implementing decorator stacks and controlling the order of execution is important when multiple decorators are applied to a class or class method. The order of execution determines how the decorators wrap and modify the behavior of the underlying code.

Example

```
from functools import wraps

def log_entry_exit(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f'Entering {func.__name__}')
        result = func(*args, **kwargs)
        print(f'Exiting {func.__name__}')
        return result
    return wrapper

def log_exception(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        try:
            result = func(*args, **kwargs)
        except ZeroDivisionError:
            print(f'Division by zero occurred in {func.__name__}')
            result = None
        except Exception as e:
            print(f'Exception in {func.__name__}: {e}')
            raise
        return result
    return wrapper

class Calculator:
    def __init__(self):
        self.result = 0

    @log_entry_exit
    def add(self, a, b):
        self.result = a + b
        return self.result

    @log_entry_exit
    @log_exception
    def divide(self, a, b):
        self.result = a / b
        return self.result

calc = Calculator()

calc.add(5, 3)  # Output: Entering add, Exiting add
calc.divide(10, 0)  # Output: Entering divide, Division by zero occurred in divide
```

In this example, we have a `Calculator` class that performs basic mathematical operations. We have two decorators, `log_entry_exit` and `log_exception`, which enhance the behavior of the class methods.

The `log_entry_exit` decorator logs when a method is entered and exited. It wraps the method and adds logging statements before and after the method call.

The `log_exception` decorator logs any exceptions that occur within the method. It wraps the method with a try-except block and prints the exception message if an exception occurs. It then re-raises the exception to propagate it further.

By applying the decorators to the class methods, we can automatically log method entry and exit, as well as handle exceptions in a centralized manner. This can be useful for debugging, monitoring, and maintaining the system.

This example demonstrates how decorator stacking can be used to enhance class behavior and add additional functionality to methods. By combining multiple decorators, we can create a powerful and flexible system for logging and exception handling.

3.6 Creating Decorator Classes with _call_ Method

To create a decorator class with the `__call__` method, you can define a class and implement the `__call__` method inside it. The `__call__` method allows the instance of the class to be called as a function.

Here's an example:

```
class DecoratorClass:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        # Code to be executed before calling the decorated function
        print("DecoratorClass: Before function call")

        # Call the decorated function
        result = self.func(*args, **kwargs)

        # Code to be executed after calling the decorated function
        print("DecoratorClass: After function call")

        return result

@DecoratorClass
def decorated_function():
    print("Inside decorated_function")

# Call the decorated function
decorated_function()
```

In this example, the `DecoratorClass` is defined with the `__call__` method. The `__init__` method takes the function to be decorated as an argument and stores it as an instance variable (`self.func`). The `__call__` method is invoked when the decorated function is called. It executes the code before and after calling the decorated function.

The `decorated_function` is decorated using the `@DecoratorClass` syntax. When `decorated_function` is called, the `__call__` method of the `DecoratorClass` instance is invoked, and the before and after code is executed.

The output of the example will be:

```
DecoratorClass: Before function call
Inside decorated_function
DecoratorClass: After function call
```

This demonstrates how to create a decorator class using the `__call__` method, allowing you to customize the behavior before and after calling the decorated function.

3.7 Class-level Decorators vs. Instance-level Decorators

Class-level decorators and instance-level decorators differ in the scope at which they are applied and the behavior they exhibit.

1. **Class-level decorators:**

- Applied to the entire class rather than specific instances.
- Decorate the class itself and affect all instances of the class.
- Typically defined using the `@decorator` syntax directly above the class definition.
- Class-level decorators are executed once when the class is defined, not when instances are created.
- They can be used to modify class attributes, class methods, or class behavior.

2. **Instance-level decorators:**

- Applied to specific instances of a class.
- Decorate individual instances of the class, allowing customization on a per-instance basis.
- Typically defined as methods within the class, and they are applied to specific instances by calling them.

- Instance-level decorators are executed each time the decorated method is called on the instance.
- They can be used to modify instance-specific attributes, perform additional checks, or add functionality to specific instances.

Here's an example to illustrate the difference:

```
class ClassDecorator:
    def __init__(self, cls):
        self.cls = cls

    def __call__(self, *args, **kwargs):
        print("ClassDecorator: Before class instantiation")
        instance = self.cls(*args, **kwargs)
        print("ClassDecorator: After class instantiation")
        return instance

@ClassDecorator
class MyClass:
    def __init__(self, value):
        self.value = value

    def method(self):
        print(f"MyClass: Method called with value {self.value}")

my_instance = MyClass(10)
my_instance.method()
```

In this example, `ClassDecorator` is a class-level decorator that wraps the instantiation of `MyClass`. When the `MyClass` class is defined, the `ClassDecorator` is executed, and the class instantiation is modified accordingly. The output will be:

```
ClassDecorator: Before class instantiation
ClassDecorator: After class instantiation
MyClass: Method called with value 10
```

As you can see, the class-level decorator is invoked once during class definition, affecting all instances of `MyClass`. It performs actions before and after class instantiation.

3.9 Decorator Class Factories: Generating Decorator Instances

Decorator class factories are a way to dynamically generate decorator instances based on input parameters or configurations. This allows for more flexibility in creating decorators with different behaviors or configurations.

Here's an example of a decorator class factory:

```
from functools import wraps

def decorator_factory(config):
    class Decorator:
        def __init__(self, func):
            self.func = func
            self.config = config

        def __call__(self, *args, **kwargs):
            # Decorator logic using self.config
            print(f"Decorator config: {self.config}")
            return self.func(*args, **kwargs)

    return Decorator

# Creating decorator instances with different configurations
decorator1 = decorator_factory(config='config1')
decorator2 = decorator_factory(config='config2')

@decorator1
def my_function():
    print("My function")

@decorator2
def another_function():
    print("Another function")

# Invoking the decorated functions
my_function()
another_function()
```

In this example, `decorator_factory` is a function that returns a decorator class based on the given configuration. The decorator class, `Decorator`, is defined within the factory function and takes the function to be decorated as an argument during initialization. The decorator class implements the `__call__` method to define the decorator's behavior. The configuration is stored as an attribute of the decorator instance.

By invoking `decorator_factory` with different configurations, we can generate different decorator instances. These instances can be used to decorate functions, and the corresponding configuration will be applied when the decorated functions are called.

The output will be:

```
Decorator config: config1
My function
Decorator config: config2
Another function
```

As you can see, the decorator instances `decorator1` and `decorator2` have different configurations, and their respective configurations are printed when the decorated functions are called.

3.10 Advanced Techniques with Decorator Classes

Advanced techniques with decorator classes involve leveraging additional features and capabilities to enhance their functionality. Here are some advanced techniques that can be used with decorator classes:

1. **Accepting Arguments:** Decorator classes can be designed to accept additional arguments to customize their behavior. These arguments can be passed during initialization or through the decorator syntax. For example:

```
class CustomDecorator:
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2

    def __call__(self, func):
        # Decorator logic using self.arg1 and self.arg2
        ...

@CustomDecorator('value1', 'value2')
def my_function():
    ...
```

2. **Handling Class Decorators**: Decorator classes can be designed to handle both class-level and instance-level decorations. This can be achieved by implementing the `__call__` method for instances and the `__get__` method for class-level decorations. This allows decorators to be applied to both functions and classes.

For example:

```
class Decorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        # Instance-level decoration logic
        ...

    def __get__(self, instance, owner):
        # Class-level decoration logic
        ...

@Decorator
def my_function():
    ...

@Decorator
class MyClass:
    ...
```

3. **Preserving Metadata:** When decorating functions or methods, decorator classes can use the `functools.wraps` decorator or manually update the attributes of the decorated function to preserve important metadata such as name, docstring, and module. This ensures that the decorated function retains its original identity and properties.

For example:

```
from functools import wraps

class Decorator:
    def __init__(self, func):
        self.func = func
        wraps(func)(self)

    def __call__(self, *args, **kwargs):
        # Decorator logic
        ...

@Decorator
def my_function():
    """Original docstring"""
    ...
```

4. **Decorator Composition:** Decorator classes can be composed to create more complex behaviors by stacking or chaining them. This allows multiple decorators to be applied sequentially to a function or method. Decorator composition can be achieved by nesting decorator classes or using the `@decorator1` syntax multiple times.

For example:

```
@decorator1
@decorator2
def my_function():
    ...
```

In this case, `my_function` will be decorated first with `decorator2` and then with `decorator1`, resulting in a composition of their behaviors.

These advanced techniques demonstrate the versatility and power of decorator classes, allowing for fine-grained control over the behavior and customization of decorated functions and classes.

*** Metaprogramming and Code Generation with Decorators**

3.11 Metaprogramming vs. Metaclasses: Choosing the Right Approach

Metaprogramming and metaclasses are two approaches in Python that allow you to modify or extend the behavior of classes and objects. While they are related concepts, they serve different purposes and should be chosen based on the specific requirements of your use case. Here's a comparison of metaprogramming and metaclasses to help you choose the right approach:

Metaprogramming:
- Metaprogramming involves writing code that generates or modifies code dynamically at runtime. It allows you to programmatically create classes, functions, and other code constructs.
- Metaprogramming techniques include using decorators, function decorators, class decorators, and using built-in functions like `setattr()` and `getattr()`.
- Metaprogramming is useful for adding new methods, attributes, or functionality to existing classes, modifying class behavior, or generating code dynamically based on certain conditions.
- Metaprogramming is more flexible and can be applied to individual instances or subclasses of a class.
- It's important to use metaprogramming judiciously as it can make code more complex and harder to understand and maintain.

Metaclasses:
- Metaclasses are a mechanism in Python that allows you to define the behavior of classes themselves. They are used to create new classes dynamically.
- Metaclasses are defined by creating a subclass of the built-in `type` class and overriding its methods, such as `__new__` and `__init__`.
- Metaclasses provide a way to control how classes are created, allowing you to modify class attributes, methods, inheritance, and other class-level behavior.
- Metaclasses are useful when you need to enforce certain patterns or constraints on classes, perform automatic class transformations, or implement class-level functionality that applies to all instances of a class.
- Metaclasses are applied at the time of defining a class and affect all instances of that class.

Choosing the Right Approach:
- Use metaprogramming when you want to modify or extend the behavior of individual instances or subclasses of a class.
- Use metaclasses when you want to define the behavior of classes themselves, enforce patterns or constraints on classes, or perform class-level transformations.
- Consider the complexity and maintainability of your code when deciding to use metaprogramming or metaclasses. Metaprogramming is more flexible but can make the code more complex and harder to understand.

Example

```
class RouteMeta(type):
    def __new__(mcs, name, bases, attrs):
        # Check if the class has a `url` attribute
        if 'url' in attrs:
            # Generate a route handler function based on the class's `url` attribute
            def route_handler():
                # Code to handle the route
                print(f"Handling route: {attrs['url']}")

            # Assign the route handler function to the class's `handle` attribute
            attrs['handle'] = route_handler

        # Create the class using the modified attributes
        return super().__new__(mcs, name, bases, attrs)

# Define a base class for route handlers
class RouteHandler(metaclass=RouteMeta):
    pass

# Define specific route handler classes
class HomePageHandler(RouteHandler):
    url = '/home'

class AboutPageHandler(RouteHandler):
    url = '/about'

# Usage
HomePageHandler.handle()  # Output: Handling route: /home
AboutPageHandler.handle()  # Output: Handling route: /about
```

In this example, we define a metaclass called `RouteMeta` that intercepts the creation of classes derived from `RouteHandler`. If a derived class has a `url` attribute, the metaclass dynamically generates a `handle` method for that class. The `handle` method is responsible for handling the route specified by the `url` attribute.

We then define two specific route handler classes, `HomePageHandler` and `AboutPageHandler`, which inherit from `RouteHandler` and define their respective `url` attributes. When we call the `handle` method on these classes, it executes the dynamically generated route handler function.

Note that this is a simplified example to illustrate the concept of metaprogramming and metaclasses in a web framework context. In real-world scenarios, web frameworks like Django or Flask provide more sophisticated mechanisms for handling routes and request/response processing.

In general, metaprogramming is more commonly used and provides enough flexibility for most use cases. Metaclasses are more advanced and should be used when you need fine-grained control over class creation and behavior.

3.12 Code Generation and Modification with Decorators

Code generation and modification with decorators can be a powerful technique for automating repetitive tasks or extending the functionality of existing code.

Here's an example that demonstrates code modification using decorators:

```
def uppercase_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        if isinstance(result, str):
            return result.upper()
        return result
    return wrapper

def append_decorator(suffix):
    def decorator(func):
        def wrapper(*args, **kwargs):
            result = func(*args, **kwargs)
            if isinstance(result, str):
                return result + suffix
            return result
        return wrapper
    return decorator

@uppercase_decorator
def greet(name):
    return f"Hello, {name}!"

@append_decorator("!!!")
def exclaim(message):
    return message

print(greet("John"))  # Output: HELLO, JOHN!
print(exclaim("This is awesome"))  # Output: This is awesome!!!
```

In this example, we have two decorators: `uppercase_decorator` and `append_decorator`. The `uppercase_decorator` decorates the `greet` function, which converts the returned string to uppercase. The `append_decorator` decorates the `exclaim` function and appends a given suffix to the returned string.

When we call `greet("John")`, the `greet` function is executed, and the returned string is automatically converted to uppercase by the `uppercase_decorator`.

Similarly, when we call `exclaim("This is awesome")`, the `exclaim` function is executed, and the returned string is automatically appended with the specified suffix "!!!" by the `append_decorator`.

By applying these decorators, we can modify the behavior of the decorated functions without explicitly modifying their implementation. This approach allows for flexible code customization and reuse.

3.13 Advanced Metaprogramming Techniques with Decorators

Advanced metaprogramming techniques with decorators involve manipulating the code at runtime, introspecting the code structure, and dynamically generating or modifying code based on certain conditions or requirements. Here are some advanced metaprogramming techniques that can be achieved using decorators:

1. **Code Injection**: Decorators can be used to inject additional code before or after the execution of a function or class. This technique is commonly used for logging, performance monitoring, or adding behavior to existing code without modifying its implementation.

2. **Attribute Modification:** Decorators can modify attributes of functions or classes dynamically. This can include adding, removing, or modifying attributes based on certain conditions or criteria. This technique is often used for extending the behavior of functions or classes at runtime.

3. **Function Wrapping**: Decorators can wrap a function with additional functionality by defining a wrapper function that calls the original function. This allows you to intercept and modify the inputs, outputs, or behavior of the original function.

4. **Class Generation:** Decorators can generate new classes dynamically by modifying the class definition or generating a new class based on certain criteria. This technique is useful for creating class hierarchies, mixins, or dynamically generating classes based on configuration or input.

5. **Metaclass Manipulation**: Decorators can be applied to metaclasses to modify the behavior of class creation. Metaclasses are responsible for creating classes, and by decorating a metaclass, you can customize how classes are generated or modified.

6. **Dynamic Code Generation:** Decorators can dynamically generate code or modify existing code structures based on specific conditions. This can include generating new functions, modifying control flow, or dynamically altering the code structure at runtime.

```
def cache_result(func):
    cache = {}

    def wrapper(*args, **kwargs):
        key = (func, args, frozenset(kwargs.items()))
        if key in cache:
            return cache[key]
        result = func(*args, **kwargs)
        cache[key] = result
        return result

    return wrapper

def cache_class_methods(cls):
    for name, value in vars(cls).items():
        if callable(value):
            setattr(cls, name, cache_result(value))
    return cls

@cache_class_methods
class MathUtils:
    @staticmethod
    def fibonacci(n):
        if n <= 1:
            return n
        return MathUtils.fibonacci(n - 1) + MathUtils.fibonacci(n - 2)

    @staticmethod
    def factorial(n):
        if n == 0:
            return 1
        return n * MathUtils.factorial(n - 1)

fib = MathUtils.fibonacci(10)  # Fibonacci sequence is cached
factorial = MathUtils.factorial(5)  # Factorial calculation is cached
```

In this example, we have two decorators: `cache_result` and `cache_class_methods`.

The `cache_result` decorator is used to cache the results of a function by storing them in a dictionary. The decorator checks if the function has been called with the same arguments before, and if so, returns the cached result instead of recomputing it.

The `cache_class_methods` decorator is applied to the `MathUtils` class. It loops through all the attributes of the class and applies the `cache_result` decorator to each callable attribute (i.e., class methods). This decorator effectively caches the results of all class methods of `MathUtils` by wrapping them with the `cache_result` decorator.

By applying the `cache_class_methods` decorator to the `MathUtils` class, the Fibonacci and factorial calculations performed by the class methods are cached. Subsequent calls to these methods with the same arguments will return the cached results instead of recomputing them.

This real-life application demonstrates how decorators and metaprogramming techniques can be combined to enhance the behavior of a class by adding caching functionality to its methods. This can significantly improve performance by avoiding redundant computations when the same inputs are provided to the methods.

These advanced metaprogramming techniques allow for powerful code customization and automation. However, they should be used with caution as they can make the code more complex and harder to understand. It's important to strike a balance between code flexibility and maintainability when using advanced metaprogramming techniques with decorators.

3.14 Building Domain-Specific Languages (DSLs) with Decorators

Building Domain-Specific Languages (DSLs) with decorators is a powerful technique that allows you to create a custom language or syntax specific to a particular domain or problem space. Decorators can be used to modify the behavior of functions or classes, allowing you to define new syntax and semantics.

Here's a simplified example of building a DSL for HTML generation using decorators:

```
def html_element(tag):
    def decorator(func):
        def wrapper(*args, **kwargs):
            content = func(*args, **kwargs)
            attributes = ' '.join([f'{k}="{v}"' for k, v in kwargs.items()])
            return f'<{tag} {attributes}>{content}</{tag}>'
        return wrapper
    return decorator

@html_element('div')
def greeting(name):
    return f'Hello, {name}!'

@html_element('p')
def introduction():
    return 'Welcome to our website.'

html = greeting('John')
html += introduction()
print(html)
```

In this example, the `html_element` decorator is defined as a decorator factory. It takes a `tag` argument and returns a decorator. The returned decorator modifies the behavior of the decorated function by wrapping it with additional HTML tags.

The `greeting` and `introduction` functions are decorated with `html_element`, which provides the desired HTML tag for each function. When these functions are called, they return the content that should be placed inside the HTML tags.

The resulting HTML is generated by calling the decorated functions and concatenating their output. The final HTML output is `<div>Hello, John!</div><p>Welcome to our website.</p>`.

This example demonstrates how decorators can be used to create a DSL for generating HTML code. By applying the `html_element` decorator to functions, we can define custom syntax and semantics specific to HTML generation.

Example

Example using decorators to build a Domain-Specific Language (DSL) for an Electronic Medical Records (EMR) system for nurses.

In this example, we'll focus on creating decorators that handle authentication and authorization for accessing patient records in the EMR system.

```
def is_nurse_authenticated():
    # Placeholder function to check nurse authentication status
    return True  # Replace with actual authentication logic

def is_nurse_authorized(role):
    # Placeholder function to check nurse authorization based on role
    return True  # Replace with actual authorization logic

def authenticate(func):
    def wrapper(*args, **kwargs):
        # Check nurse's authentication status
```

```
            if not is_nurse_authenticated():
                raise PermissionError("Nurse authentication failed")
            return func(*args, **kwargs)
        return wrapper

    def authorize(role):
        def decorator(func):
            def wrapper(*args, **kwargs):
                # Check nurse's authorization based on role
                if not is_nurse_authorized(role):
                    raise PermissionError("Nurse is not authorized")
                return func(*args, **kwargs)
            return wrapper
        return decorator

    class EMRSystem:
        @authenticate
        @authorize("nurse")
        def view_patient_record(self, patient_id):
            # Retrieve patient record based on patient_id
            patient_record = self.get_patient_data(patient_id)
            return patient_record

        @authenticate
        @authorize("admin")
        def add_patient_record(self, patient_data):
            # Add new patient record to the system
            # Only admin nurses can perform this action
            self.add_patient_data(patient_data)
            return "Patient record added successfully"

        def get_patient_data(self, patient_id):
            # Retrieve patient data based on patient_id
            # Replace with your actual implementation
            # Example implementation:
            # patient_data = database.query("SELECT * FROM patients WHERE id = ?", patient_id)
            # return patient_data
            pass

        def add_patient_data(self, patient_data):
            # Add new patient data to the system
            # Replace with your actual implementation
            # Example implementation:
            # database.insert(patient_data)
            pass

    # Usage example
    emr = EMRSystem()

    # Nurse trying to view patient record
    try:
        patient_record = emr.view_patient_record("123")
        print(patient_record)
    except PermissionError as e:
        print(f"Access Denied: {str(e)}")

    # Admin nurse trying to add patient record
    try:
        patient_data = {"patient_id": "456", "name": "John Doe", "age": 35}
        response = emr.add_patient_record(patient_data)
        print(response)
    except PermissionError as e:
        print(f"Access Denied: {str(e)}")
```

In this example, the `authenticate` decorator checks whether the nurse is authenticated before executing the decorated method. If the nurse is not authenticated, a `PermissionError` is raised.

The `authorize` decorator takes a role as an argument and checks if the nurse is authorized based on that role. If the nurse is not authorized, a `PermissionError` is raised.

The `EMRSystem` class has two methods: `view_patient_record` and `add_patient_record`. These methods are decorated with `authenticate` and `authorize` decorators to ensure proper authentication and authorization checks before accessing patient records.

In a real-world application, the EMR system for nurses would have more functionality and additional decorators to handle various aspects such as data validation, logging, error handling, and more. The decorators provide a clean and reusable way to enforce security and behavior rules within the EMR system, making the code more maintainable and secure

4. Common Built-in Decorators

Python provides several built-in decorators that are commonly used in various scenarios. Here are some of the most common built-in decorators:

1. `@property`: This decorator is used to define a method as a property of a class. It allows you to access the method like an attribute without using parentheses.

```
class Person:
    def __init__(self, name):
```

```
        self._name = name

    @property
    def name(self):
        return self._name

person = Person("John")
print(person.name)  # Access the name property without parentheses
```

2. `@staticmethod` : This decorator is used to define a static method within a class. Static methods do not have access to the instance or class attributes and can be called on the class itself.

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

result = MathUtils.add(2, 3)  # Call the static method on the class
```

3. `@classmethod` : This decorator is used to define a class method within a class. Class methods receive the class itself as the first argument and can be called on both the class and instances of the class.

```
class MathUtils:
    @classmethod
    def multiply(cls, a, b):
        return a * b

result1 = MathUtils.multiply(2, 3)  # Call the class method on the class
result2 = MathUtils().multiply(2, 3)  # Call the class method on an instance
```

4. `@abstractmethod` : This decorator is used to define an abstract method within an abstract base class (ABC). Abstract methods do not have an implementation and must be overridden by the subclasses.

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2
```

5. **@cached_property and @lru_cache**

The `@cached_property` and `@lru_cache` decorators are built-in decorators in Python that provide caching functionality for properties and function calls, respectively.

`@cached_property` : The `@cached_property` decorator can be used to cache the value of a property. It ensures that the property is computed only once and then cached for subsequent access. This can be useful when the computation of a property is expensive and you want to avoid recomputing it multiple times.

Here's an example of how to use the `@cached_property` decorator:

```
from functools import cached_property

class MyClass:
    @cached_property
    def expensive_property(self):
        # Perform expensive computation
        result = ...
        return result
```

In the example above, the `expensive_property` is decorated with `@cached_property`. The first time the property is accessed, the computation is performed and the result is cached. On subsequent access, the cached value is returned instead of recomputing the property.

`@lru_cache`: The `@lru_cache` decorator (Least Recently Used cache) can be used to cache the results of function calls. It stores the most recent function calls and their results in a cache, allowing for faster subsequent calls with the same arguments.

Here's an example of how to use the `@lru_cache` decorator:

```
from functools import lru_cache

@lru_cache
def expensive_function(arg):
    # Perform expensive computation
    result = ...
    return result
```

4.1 Preserving Function Metadata with functools.wraps

When creating decorators in Python, it's important to preserve the metadata of the original function, such as its name, docstring, and module, to maintain consistency and readability. The `functools.wraps` decorator can be used to achieve this.

The `functools.wraps` decorator is a decorator itself that is designed specifically for creating other decorators. It copies the metadata from the original function to the decorated function, ensuring that important information is preserved. It updates the decorated function with attributes from the original function, including `__name__`, `__doc__`, `__module__`, and others.

Here's an example that demonstrates how to use `functools.wraps` to preserve function metadata:

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Decorator logic
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def my_function():
    """This is my function."""
    pass

print(my_function.__name__)  # Output: my_function
print(my_function.__doc__)   # Output: This is my function.
print(my_function.__module__) # Output: __main__
```

In the example above, the `my_decorator` function is decorated with `@wraps(func)`. This ensures that the metadata of the original function passed to `my_decorator` is copied over to the `wrapper` function. When `my_function` is decorated with `@my_decorator`, the metadata of `my_function` is preserved, including the name, docstring, and module.

Example

```
from functools import wraps
from flask import Flask, request, jsonify

app = Flask(__name__)

# Placeholder functions for authentication and authorization
def is_nurse_authenticated():
    # Logic to check nurse's authentication
    pass

def is_nurse_authorized(permissions):
    # Logic to check nurse's authorization based on permissions
    pass

# Placeholder function to retrieve patient data
def get_patient_data(patient_id):
    # Logic to retrieve patient data
    pass

# Decorator to authenticate nurses
def authenticate_nurse(func):
    @wraps(func)
```

```
    def wrapper(*args, **kwargs):
        # Check nurse's authentication credentials
        if not is_nurse_authenticated():
            return jsonify({'message': 'Authentication failed'}), 401
        return func(*args, **kwargs)
    return wrapper

# Decorator to authorize nurses for specific actions
def authorize_nurse(permissions):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Check nurse's authorization based on permissions
            if not is_nurse_authorized(permissions):
                return jsonify({'message': 'Unauthorized'}), 403
            return func(*args, **kwargs)
        return wrapper
    return decorator

# Example usage of decorators in an EMR system for nurses

@app.route('/patient/<patient_id>')
@authenticate_nurse
@authorize_nurse(['view_patient'])
def view_patient(patient_id):
    """
    Retrieve and display patient information.

    :param patient_id: The ID of the patient.
    :return: JSON response containing patient data.
    """
    patient_data = get_patient_data(patient_id)
    return jsonify({'patient': patient_data}), 200

# Other route handlers and functionality...

if __name__ == '__main__':
    app.run()
```

In this updated example, the placeholder functions `is_nurse_authenticated`, `is_nurse_authorized`, and `get_patient_data` are expected to be implemented with the actual authentication, authorization, and data retrieval logic suitable for the EMR system.

By integrating these functions into the decorators, you can achieve authentication and authorization checks before allowing access to the protected routes. Additionally, the `wraps` decorator ensures that the decorated function retains its original metadata, making it easier to understand and work with in the EMR system.

By using `functools.wraps`, you can create decorators that seamlessly integrate with the decorated functions, preserving their original metadata. This helps maintain consistency and clarity when working with decorated functions, especially in scenarios where introspection or documentation generation is involved.

4.2 Context Managers as Decorators

Context managers can also be used as decorators to provide additional behavior or resource management to functions or methods. When a context manager is used as a decorator, it allows you to define setup and teardown logic around the execution of the decorated function or method.

Here's an example of using a context manager as a decorator:

```
from contextlib import contextmanager
from functools import wraps

@contextmanager
def timer():
    import time
    start_time = time.time()
    try:
        yield
    finally:
        end_time = time.time()
        elapsed_time = end_time - start_time
        print(f"Elapsed time: {elapsed_time} seconds")

def timer_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        with timer():
            return func(*args, **kwargs)
    return wrapper

@timer_decorator
def perform_task():
    # Functionality to be timed
    import time
    time.sleep(2)
    print("Task complete")
```

```
perform_task()
```

The `perform_task` function is decorated with `@timer`, which means the `timer` context manager will be invoked before and after the execution of `perform_task`. The code block within the `perform_task` function will be executed between the `yield` statement.

When `perform_task` is called, it will output the elapsed time for the task.

Context managers can also be used as decorators to provide additional behavior or resource management to functions or methods. When a context manager is used as a decorator, it allows you to define setup and teardown logic around the execution of the decorated function or method.

Here's an example of using a context manager as a decorator:

```
from functools import wraps

def memoize(func):
    cache = {}

    @wraps(func)
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]

    return wrapper

def cache(max_size):
    cache = {}

    def decorator(func):
        @wraps(func)
        def wrapper(*args):
            if args in cache:
                return cache[args]

            result = func(*args)
            cache[args] = result

            if len(cache) > max_size:
                cache.popitem(last=False)

            return result

        return wrapper

    return decorator
```

4.3 Decorators for Memoization and Caching

Memoization and caching are common use cases for decorators. They can significantly improve the performance of functions by storing the results of expensive computations and returning them directly for subsequent calls with the same arguments. Here's an example of decorators for memoization and caching:

```
from functools import wraps

def memoize(func):
    cache = {}

    @wraps(func)
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]

    return wrapper

def cache(max_size):
    cache = {}

    def decorator(func):
        @wraps(func)
        def wrapper(*args):
            if args in cache:
                return cache[args]

            result = func(*args)
            cache[args] = result

            if len(cache) > max_size:
                cache.popitem(last=False)

            return result
```

```
        return wrapper

    return decorator
```

The `memoize` decorator implements a simple memoization technique. It creates a cache dictionary to store the results of function calls based on their arguments. If a function is called with the same arguments, it returns the cached result instead of recomputing it.

The `cache` decorator, on the other hand, adds an additional feature of limiting the cache size. It takes a `max_size` parameter and uses a dictionary to store the results with a maximum size.

Example:

```
from functools import wraps

def memoize(func):
    cache = {}

    @wraps(func)
    def wrapper(user_id):
        if user_id not in cache:
            cache[user_id] = func(user_id)
        return cache[user_id]

    return wrapper

def cache(max_size):
    cache = {}

    def decorator(func):
        @wraps(func)
        def wrapper(user_id):
            if user_id in cache:
                return cache[user_id]

            result = func(user_id)
            cache[user_id] = result

            if len(cache) > max_size:
                cache.popitem(last=False)

            return result

        return wrapper

    return decorator

@cache(max_size=1000)
def generate_recommendations(user_id):
    # Expensive computation to generate recommendations
    recommendations = compute_recommendations(user_id)
    # ...

    return recommendations

def compute_recommendations(user_id):
    # Placeholder implementation
    # Replace with actual recommendation generation logic
    recommendations = ['Recommendation 1', 'Recommendation 2', 'Recommendation 3']
    return recommendations

# Usage example
user_id = 123
recommendations = generate_recommendations(user_id)
```

the `compute_recommendations` function generates a list of recommendations as a placeholder implementation. Replace this function with your actual logic for generating recommendations based on the provided `user_id`.

After replacing the placeholder logic, you can call the `generate_recommendations` function with a specific `user_id` to generate recommendations. The function will use memoization and caching to store and retrieve the results, improving performance by avoiding redundant computations.

Make sure to customize the `compute_recommendations` function according to your application's requirements to provide meaningful recommendations for the given `user_id`.

4.4 Method and Attribute Decorators

Method and attribute decorators are used to modify or enhance the behavior of methods and attributes in classes. They can be applied to individual methods or attributes within a class, allowing you to add additional functionality or perform custom operations.

Here's an example of method and attribute decorators:

```
def log_calls(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

class MyClass:
    @log_calls
    def my_method(self, x, y):
        return x + y

    @property
    @log_calls
    def my_attribute(self):
        return "Hello, world!"

obj = MyClass()
result = obj.my_method(3, 4)
print(result)  # Output: Calling function: my_method, 7

value = obj.my_attribute
print(value)  # Output: Calling function: my_attribute, Hello, world!
```

In this example, the `log_calls` decorator is defined to print a log message before executing a method or accessing an attribute. It wraps the original method or attribute, allowing you to add custom behavior before and after the original code execution.

The `log_calls` decorator is applied to both the `my_method` method and the `my_attribute` property in the `MyClass` class. When the `my_method` method is called, the decorator prints a log message before executing the original code. Similarly, when the `my_attribute` property is accessed, the decorator prints a log message before returning the attribute value.

Decorators provide a powerful way to modify or extend the behavior of methods and attributes in classes. They can be used for various purposes such as logging, input validation, performance monitoring, access control, and more.

4.5 Leveraging Decorators for Debugging and Profiling

Decorators can be leveraged for debugging and profiling purposes to gain insights into the execution of functions and methods. They allow you to add debugging statements, track execution time, and collect other relevant information during runtime.

Here's an example of using decorators for debugging and profiling:

```
import time

def debug(func):
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        print(f"Arguments: {args}")
        print(f"Keyword arguments: {kwargs}")
        result = func(*args, **kwargs)
        print(f"Result: {result}")
        return result
    return wrapper

def profile(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Function: {func.__name__}")
        print(f"Execution time: {execution_time} seconds")
        return result
    return wrapper

@debug
@profile
def my_function(x, y):
    # Perform some computation
    result = x + y
    return result

# Usage example
result = my_function(3, 4)
print(result)  # Output: 7
```

In this example, the `debug` decorator adds debug statements to print information about the function being called, its arguments, and the returned result. The `profile` decorator measures the execution time of the function and prints it.

The `debug` decorator is applied first, followed by the `profile` decorator, using multiple decorator syntax. This means that the `debug` decorator wraps the `profile` decorator, and the `profile` decorator wraps the original function.

When `my_function` is called with arguments `3` and `4`, the decorators add the necessary debug and profiling statements, providing insights into the function's behavior and execution time.

Decorators for debugging and profiling can be useful for understanding the flow of code, identifying bottlenecks, and optimizing performance. They offer a convenient way to add diagnostic information without modifying the original code.

*** Decorating Generators and Coroutines**
4.6 Understanding Generators and Coroutines

Generators and coroutines are powerful concepts in Python that allow for the creation of iterable sequences and asynchronous programming, respectively. Let's explore each concept briefly:

1. **Generators:**

- Generators are functions that can be paused and resumed during their execution, allowing for the generation of a sequence of values over time.
- They are defined using the `yield` keyword instead of the `return` keyword. When a generator function is called, it returns an iterator object.
- Each time the `yield` statement is encountered, the generator function pauses and yields a value to the caller. The state of the function is saved, allowing it to be resumed later.
- Generators are memory-efficient as they generate values on-the-fly, one at a time, rather than generating them all at once.
- They are commonly used for iterating over large or infinite sequences, processing streaming data, and implementing custom iterators.

Example:

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

fib_gen = fibonacci()
print(next(fib_gen))  # 0
print(next(fib_gen))  # 1
print(next(fib_gen))  # 1
print(next(fib_gen))  # 2
# ...
```

Coroutines:

- Coroutines are a more advanced concept than generators. They are designed for asynchronous programming and cooperative multitasking.
- Coroutines are defined using the `async def` syntax. When a coroutine function is called, it returns a coroutine object.
- Coroutines use the `await` keyword to pause their execution and wait for other coroutines or asynchronous operations to complete.
- Coroutines can be scheduled and executed concurrently, allowing for efficient and non-blocking execution of multiple tasks.
- They are commonly used in asynchronous frameworks like asyncio, where they enable asynchronous I/O, concurrent programming, and handling of long-running operations without blocking the event loop.

Example:

```
import asyncio

async def fetch_data(url):
    # Simulate asynchronous data fetching
    await asyncio.sleep(1)
    return "Data from " + url

async def process_data():
    url = "https://example.com"
    data = await fetch_data(url)
    print("Received data:", data)

asyncio.run(process_data())
```

Example

```
import asyncio

async def fetch_patient_data(patient_id):
    # Simulate fetching patient data from a remote server
    await asyncio.sleep(1)
    return {"patient_id": patient_id, "name": "John Doe", "age": 35}

async def update_patient_record(patient_id, record_data):
    # Simulate updating the patient record
    await asyncio.sleep(0.5)
    print(f"Updated patient record for patient {patient_id}: {record_data}")

async def send_notification(patient_id, message):
    # Simulate sending a notification to healthcare providers
    await asyncio.sleep(0.2)
    print(f"Notification sent for patient {patient_id}: {message}")

async def nurse_workflow(patient_id):
    # Fetch patient data
    patient_data = await fetch_patient_data(patient_id)
    print(f"Fetched patient data: {patient_data}")

    # Update patient record
    await update_patient_record(patient_id, {"diagnosis": "Fever", "treatment": "Antipyretics"})

    # Send notification to healthcare providers
    await send_notification(patient_id, "Patient requires further evaluation")

asyncio.run(nurse_workflow("P001"))
```

In this example, we define several coroutines: `fetch_patient_data`, `update_patient_record`, and `send_notification`. Each coroutine simulates an asynchronous operation with a delay using `asyncio.sleep`.

The `nurse_workflow` coroutine represents a typical workflow that a nurse might follow. It uses `await` to asynchronously call other coroutines, such as fetching patient data, updating the patient record, and sending notifications.

By leveraging coroutines and `asyncio`, the nurse can initiate multiple workflows concurrently without waiting for each operation to complete. This allows for better responsiveness and efficiency in handling EMR tasks.

Please note that this is a simplified example to demonstrate the concept. In a real-world application, you would have more complex logic and integration with EMR systems, databases, and external services.

4.7 Decorating Generator Functions

Decorating generator functions allows us to modify or enhance the behavior of the generator by wrapping it with additional functionality. We can achieve this by creating a decorator that takes a generator function as input and returns a modified version of that generator.

Here's an example of decorating a generator function:

```
def uppercase_decorator(generator_func):
    def wrapper():
        # Get the generator object
        generator = generator_func()

        # Iterate over the values produced by the generator
        for value in generator:
            # Modify the value before yielding
            yield value.upper()

    return wrapper

@uppercase_decorator
def name_generator():
    yield 'john'
    yield 'jane'
    yield 'michael'

# Using the decorated generator
for name in name_generator():
    print(name)
```

In this example, we have a generator function `name_generator()` that yields names. We create a decorator `uppercase_decorator` that wraps the generator function with additional functionality. The decorator returns a modified version of the generator that yields uppercase names.

When we use the decorator `@uppercase_decorator` above the `name_generator()` definition, it is equivalent to `name_generator = uppercase_decorator(name_generator)`. It replaces the original generator function with the decorated version.

Example

```
def validate_data(generator_func):
    def wrapper():
        generator = generator_func()

        for data in generator:
            # Perform data validation
            if validate(data):
                yield data
            else:
                print(f"Invalid data: {data}")

    return wrapper

def cache_results(generator_func):
    cache = {}

    def wrapper():
        generator = generator_func()

        for item in generator:
            key = item['id']
            if key in cache:
                yield cache[key]
            else:
                result = process_data(item)
                cache[key] = result
                yield result

    return wrapper

@validate_data
@cache_results
def stream_data():
    for item in external_data_source():
        yield item

# Simulated external data source
def external_data_source():
    yield {'id': 1, 'value': 10}
    yield {'id': 2, 'value': 20}
    yield {'id': 3, 'value': 30}
    yield {'id': 4, 'value': 'invalid'}

def validate(data):
    # Perform data validation logic
    return isinstance(data, dict) and isinstance(data.get('value'), int)


def process_data(item):
    # Simulated data processing logic
    return item['value'] * 2

# Using the decorated generator for stream processing
for result in stream_data():
    print(result)
```

In this example, we have a stream processing scenario where data is coming from an external source. We want to validate the data and cache the processed results to optimize performance.

The `validate_data` decorator validates each item yielded by the generator, filtering out any invalid data. The `cache_results` decorator caches the processed results to avoid recomputing them for duplicate inputs.

The `stream_data` generator is decorated with `@validate_data` and `@cache_results`, which apply the decorators in the specified order. This means the data is first validated and then processed, with caching applied to avoid redundant processing.

The `external_data_source` simulates the external source providing data, and the `validate` function performs the data validation logic. The `process_data` function represents the processing step for each item.

When we iterate over the `stream_data()` generator, each item goes through the validation, caching, and processing steps, and the final result is printed. Invalid data is filtered out and flagged as such.

This example demonstrates how decorators can enhance a generator-based stream processing pipeline by adding functionality like data validation and result caching. You can extend this concept by incorporating additional decorators or customizing the decorators to suit your specific requirements in an EMR application for nurses or any other real-world use case.

4.9 Applying Design Patterns with Decorators

Design patterns provide reusable solutions to common problems in software design. Decorators can be used to apply various design patterns and enhance the functionality and behavior of objects. Here are a few design patterns that can be implemented using decorators:

1. **Decorator Pattern:** The decorator pattern allows adding new behaviors or functionalities to an object dynamically without affecting its underlying structure. Decorators wrap the object and provide additional features. This pattern is useful when you want to add functionality to an object at runtime. Decorators can be used to implement features like logging, caching, or encryption.

2. **Adapter Pattern**: The adapter pattern allows objects with incompatible interfaces to work together by creating an adapter that acts as a bridge between them. Decorators can be used to adapt the interface of an object to match the interface expected by the client. This allows the client to work with the adapted object seamlessly.

3. **Composite Pattern:** The composite pattern allows treating individual objects and groups of objects uniformly. Decorators can be used to add additional functionality to individual objects or groups of objects in a composite structure. For example, you can use decorators to add behavior to specific components in a tree-like structure of objects.

4. **Proxy Pattern:** The proxy pattern provides a surrogate or placeholder for another object to control access to it. Decorators can be used to implement proxy objects that intercept requests to the real object and add additional functionality, such as access control, caching, or lazy loading.

5. **Strategy Pattern:** The strategy pattern allows defining a family of interchangeable algorithms and selecting one dynamically at runtime. Decorators can be used to implement different strategies and wrap them around an object. This allows changing the behavior of the object by dynamically selecting a different decorator.

```
from functools import wraps


# Decorator for logging method calls
def log_method_call(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        return func(*args, **kwargs)
    return wrapper


# Decorator for measuring method execution time
def measure_execution_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        import time
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"{func.__name__} executed in {execution_time:.2f} seconds")
        return result
    return wrapper


# Example class using decorators
class DataService:
    @log_method_call
    def fetch_data(self):
        # Code to fetch data from a remote server
        print("Fetching data...")
        # Simulating some delay
        import time
        time.sleep(2)
        print("Data fetched")

    @measure_execution_time
    def process_data(self):
        # Code to process the fetched data
        print("Processing data...")
        # Simulating some processing time
        import time
        time.sleep(1)
        print("Data processed")


# Usage example
data_service = DataService()
data_service.fetch_data()
data_service.process_data()
```

In this example, the `log_method_call` decorator is applied to the `fetch_data` method, which logs each method call. The `measure_execution_time` decorator is applied to the `process_data` method, which measures the execution time of the method and prints it.

By using these decorators, we enhance the functionality of the `DataService` class without modifying its underlying structure. The decorators provide additional behavior without cluttering the main code, making it more maintainable and modular.

By applying these design patterns with decorators, you can enhance the flexibility, modularity, and extensibility of your code. Decorators provide a clean and concise way to implement these patterns and separate concerns, making your code more maintainable and reusable.

4.12 Decorator Pattern vs. Decorators in Python

The Decorator Pattern and decorators in Python are related concepts but serve different purposes.

1. **Decorator Pattern**: The Decorator Pattern is a design pattern in software engineering that allows behavior to be added to an individual object dynamically, without affecting the behavior of other objects of the same class. It promotes the principle of open-closed design, where classes are open for extension but closed for modification.

The Decorator Pattern involves creating a decorator class that wraps the original object and adds additional functionality to it. The decorator class implements the same interface as the original object, allowing it to be used interchangeably. The decorators can be stacked, allowing multiple layers of behavior to be added to the object.

2. **Decorators in Python**: Decorators in Python are a language feature that allows you to modify the behavior of a function or a class by wrapping it with another function or class. Decorators are written using the `@decorator_name` syntax and are applied to the target function or class.

Decorators in Python provide a convenient way to add functionality to functions or classes without modifying their source code. They are used for various purposes such as logging, caching, authentication, validation, and more. Decorators in Python are versatile and can be applied to functions, methods, and classes.

While the Decorator Pattern is a design pattern that involves creating decorator classes, decorators in Python are a language feature that allows you to modify the behavior of functions and classes using syntactic sugar.

In summary, the Decorator Pattern is a design pattern that involves creating decorator classes to add behavior to objects dynamically, while decorators in Python are a language feature that allows you to modify the behavior of functions and classes using syntactic sugar.

4.13 Decorators for Behavioral Design Patterns

Behavioral design patterns focus on the interactions and communication between objects to provide solutions for common design problems. While decorators in Python may not directly correspond to all behavioral design patterns, they can still be used to achieve similar effects and add behavioral aspects to objects. Here are a few examples:

1. **Observer Pattern:** The Observer Pattern defines a one-to-many dependency between objects, where changes in one object trigger updates in other dependent objects. Decorators can be used to add observation capabilities to objects by wrapping them with decorators that notify observers when certain events occur.

2. **Strategy Pattern:** The Strategy Pattern defines a family of interchangeable algorithms and encapsulates each algorithm within its own class. Decorators can be used to dynamically change the behavior of an object by wrapping it with different strategies. This allows you to switch or modify the behavior of an object at runtime by applying different decorators.

3. **Chain of Responsibility Pattern**: The Chain of Responsibility Pattern establishes a chain of objects, where each object has the ability to handle a request or pass it to the next object in the chain. Decorators can be used to extend the handling capabilities of an object by wrapping it with additional decorators that handle specific aspects of the request.

4. **State Pattern**: The State Pattern allows an object to change its behavior when its internal state changes. Decorators can be used to modify the behavior of an object based on its current state by applying state-specific decorators. This allows you to dynamically alter the behavior of an object without changing its core implementation.

It's important to note that while decorators in Python can provide behavioral aspects, they may not cover all the complexities and interactions of behavioral design patterns. Behavioral patterns often involve more advanced mechanisms, such as messaging systems, state transitions, or complex interaction protocols. However, decorators can still be a valuable tool for enhancing the behavior and flexibility of objects in a Python application.

When using decorators for behavioral design patterns, carefully consider the specific requirements of the pattern and how decorators can be utilized to achieve the desired behavioral aspects in your application.

4.15 Combining Decorators and Creational Design Patterns

Combining decorators with creational design patterns can provide powerful and flexible solutions. Creational design patterns focus on object creation mechanisms, while decorators allow us to add additional behavior or modify objects dynamically.

Here's an example that combines the Singleton design pattern with a decorator:

```
def singleton(cls):
    instances = {}

    @wraps(cls)
    def wrapper(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return wrapper


@singleton
class DatabaseConnection:
    def __init__(self):
        # Initialize the database connection here
        pass

    def execute_query(self, query):
        # Execute the query
        pass

# Usage
db_connection1 = DatabaseConnection()
db_connection2 = DatabaseConnection()

print(db_connection1 is db_connection2)  # Output: True
```

In the example above, we use the `singleton` decorator to create a singleton instance of the `DatabaseConnection` class. The decorator ensures that only one instance of the class is created and reused every time the `DatabaseConnection` class is instantiated.

Example

```
def log_query(func):
    @wraps(func)
    def wrapper(self, query):
        print("Executing query:", query)
        return func(self, query)

    return wrapper

class DatabaseConnection:
    def __init__(self, host, port, username, password):
        self.host = host
        self.port = port
        self.username = username
        self.password = password

    @log_query
    def execute_query(self, query):
        # Code for executing the query
        pass

# Example usage
db_connection = DatabaseConnection('localhost', 5432, 'admin', 'password')
db_connection.execute_query("SELECT * FROM users")
```

In this example, the `singleton` decorator ensures that only one instance of the `DatabaseConnection` class is created. The `log_queries` decorator wraps the `execute_query` method of the database connection object to log the executed query before executing it.

With this approach, every query executed through the `execute_query` method of the `db_connection` object will be logged, providing valuable information for debugging and monitoring purposes.

This demonstrates how combining decorators with creational design patterns can enhance the behavior of objects and add additional functionality to real-world applications

By combining decorators with creational design patterns, you can apply additional functionality, such as caching, logging, or security, to objects created using creational patterns. This allows you to customize and enhance the behavior of objects while keeping the object creation mechanism intact.

5. **Creating Custom Decorators**

Let's explore how custom decorators can be used in a real-life application for an Electronic Medical Record (EMR) system used by nurses. We'll focus on different aspects of custom decorators.

1. **Anatomy of a Decorator Function:** The anatomy of a decorator function consists of a wrapper function that wraps the original function and performs additional functionality. In the case of an EMR application, we can create a decorator function that handles authentication and authorization before allowing access

to sensitive patient information.

2. **Decorating Functions with Custom Decorators:** Custom decorators can be used to add functionality to specific functions in the EMR application. For example, a `authenticate` decorator can be applied to functions that require authentication, ensuring that only authorized nurses can access sensitive patient data.
3. **Handling Decorator Arguments and Options**: Decorators can accept arguments to customize their behavior. In the EMR application, we can create a `role_required` decorator that takes a role argument, allowing us to specify which roles are allowed to access certain functions. This helps in managing different levels of authorization within the application.
4. **Best Practices for Writing Custom Decorators**: When writing custom decorators for the EMR application, it's important to follow best practices, such as preserving function metadata, using `functools.wraps`, and applying decorators in a consistent and logical manner. This ensures that the decorators behave as expected and maintain the integrity of the original functions.

Here's a code snippet that demonstrates these concepts in the context of an EMR application for nurses:

```
from functools import wraps

# Custom decorator for authentication
def authenticate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if is_nurse_authenticated():
            return func(*args, **kwargs)
        else:
            raise PermissionError("Access denied. Please authenticate as a nurse.")
    return wrapper

# Custom decorator for role-based authorization
def role_required(role):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            if is_nurse_authorized(role):
                return func(*args, **kwargs)
            else:
                raise PermissionError(f"Access denied. Role '{role}' required.")
        return wrapper
    return decorator

# Example usage

@authenticate
def view_patient_records(patient_id):
    # Function to view patient records
    print(f"Viewing records for patient {patient_id}")

@role_required('admin')
def delete_patient_record(patient_id):
    # Function to delete a patient record
    print(f"Deleting record for patient {patient_id}")

# Simulating authentication and authorization functions
def is_nurse_authenticated():
    # Check if the nurse is authenticated
    return True

def is_nurse_authorized(role):
    # Check if the nurse has the required role
    # Perform role-based authorization logic
    return True

# Usage of decorated functions
view_patient_records(12345)  # This will execute successfully

delete_patient_record(12345)  # This will raise a PermissionError if nurse is not authorized with 'admin' role
```

In this example, the `authenticate` decorator ensures that the nurse is authenticated before accessing patient records, and the `role_required` decorator allows only nurses with the specified role to delete patient records.

You can customize the authentication and authorization logic in the `is_nurse_authenticated` and `is_nurse_authorized` functions to fit the requirements of your EMR application.

Remember to adapt the code to your specific application requirements and integrate it with the existing EMR system used by nurses.

5.2 Timing and Benchmarking Decorators

Timing and benchmarking decorators are used to measure the execution time of functions or code blocks in order to assess their performance. They can be valuable tools for identifying bottlenecks, optimizing critical operations, and comparing different implementations.

Here's an example of how timing and benchmarking decorators can be implemented:

```
import time

# Timing decorator
def measure_execution_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Function {func.__name__} took {execution_time} seconds to execute")
        return result
    return wrapper

# Example function to measure execution time
@measure_execution_time
def my_function():
    time.sleep(2)  # Simulating a time-consuming operation

# Usage example
if __name__ == '__main__':
    my_function()
```

In the code above, we define a `measure_execution_time` decorator. This decorator wraps the target function `my_function` and measures the time it takes to execute. It uses the `time` module to calculate the elapsed time between the start and end of the function execution. The measured execution time is then printed to the console.

By applying the `measure_execution_time` decorator to a function, we can easily measure its execution time without modifying the original function code. This allows us to analyze and optimize performance as needed.

Example:

Here's an example that combines timing and benchmarking decorators with the scenario of an EMR application for nurses and the requirement of a health information dashboard:

```
import time

# Timing and benchmarking decorator
def measure_execution_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Function {func.__name__} took {execution_time} seconds to execute")
        return result
    return wrapper

# EMR Application Dashboard
class EMRDashboard:
    def __init__(self):
        self.nurse_data = {}
        self.patient_data = {}

    def fill_nurse_data(self, nurse_id, data):
        self.nurse_data[nurse_id] = data
        print(f"Nurse {nurse_id}: Health data filled")

    def fill_patient_data(self, patient_id, data):
        self.patient_data[patient_id] = data
        print(f"Patient {patient_id}: Health data filled")

# Applying decorators to measure execution time
@measure_execution_time
def process_nurse_data(nurse_dashboard, nurse_id, data):
    nurse_dashboard.fill_nurse_data(nurse_id, data)

@measure_execution_time
def process_patient_data(patient_dashboard, patient_id, data):
    patient_dashboard.fill_patient_data(patient_id, data)

# Usage example
if __name__ == '__main__':
    emr_dashboard = EMRDashboard()

    # Nurse filling health information
    process_nurse_data(emr_dashboard, "N001", {"blood_pressure": "120/80", "temperature": "98.6"})

    # Patient filling health information
    process_patient_data(emr_dashboard, "P001", {"weight": "150 lbs", "height": "5'7"})
```

In the above example, the `measure_execution_time` decorator is applied to the `process_nurse_data` and `process_patient_data` functions. It measures and logs the execution time of these functions. The `EMRDashboard` class represents the dashboard where nurses and patients can fill their health information. The decorator helps in

benchmarking the time taken to process the data in the dashboard functions.

You can apply the timing decorator to multiple functions or code blocks within your EMR application to track their execution time. This information can be helpful in identifying areas that require optimization or where performance improvements are needed.

5.6 Authentication and Authorization Decorators

Authentication and authorization decorators are commonly used in web applications to enforce access control and protect resources. They ensure that only authenticated and authorized users can access certain functionalities or perform specific actions. Here's an example of how authentication and authorization decorators can be implemented:

```
from functools import wraps

# Placeholder function for user authentication
def is_user_authenticated():
    # Implement your user authentication logic here
    # Return True if the user is authenticated, False otherwise
    pass

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if is_user_authenticated():
            return func(*args, **kwargs)
        else:
            raise Exception("User authentication failed")
    return wrapper

# Authorization decorator
def authorize(allowed_roles):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            if has_sufficient_role(allowed_roles):
                return func(*args, **kwargs)
            else:
                raise Exception("User authorization failed")
        return wrapper
    return decorator

# Example usage
@authenticate
def sensitive_operation():
    # Perform sensitive operation here
    pass

# Helper function to check if the user has sufficient role/permission
def has_sufficient_role(allowed_roles):
    # Placeholder implementation
    user_role = get_user_role()  # Implement your logic to get the user role
    return user_role in allowed_roles

# Helper function to get the role of the user
def get_user_role():
    # Placeholder implementation
    # Retrieve the role of the user from your authentication/authorization system
    return 'admin'  # Replace with actual user role retrieval logic

# Usage example
if __name__ == '__main__':
    try:
        sensitive_operation()
    except Exception as e:
        print(e)
```

In the code above, we define an `authenticate` decorator that checks if the user is authenticated before allowing access to the decorated function. If the user is authenticated, the decorated function is executed; otherwise, an exception is raised.

Example:

Here's an example implementation of authentication and authorization decorators for an EMR application that allows only nurses to perform CRUD operations on patient health information:

```
from functools import wraps

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(user_type, *args, **kwargs):
        if user_type == 'nurse' and is_user_authenticated():
            print(f"User authenticated as {user_type}")
```

```
            return func(user_type, *args, **kwargs)
        else:
            raise Exception("User authentication failed")
    return wrapper

# Authorization decorator
def authorize(func):
    @wraps(func)
    def wrapper(user_type, *args, **kwargs):
        if user_type == 'nurse' and has_sufficient_permission():
            print(f"User authorized with sufficient permission")
            return func(user_type, *args, **kwargs)
        else:
            raise Exception("User authorization failed")
    return wrapper

# Example usage
@authenticate
def add_patient_health_info(user_type, patient_id, health_info):
    print(f"Adding health information for patient {patient_id}")

@authenticate
@authorize
def modify_patient_health_info(user_type, patient_id, health_info):
    print(f"Modifying health information for patient {patient_id}")

@authenticate
def delete_patient_health_info(user_type, patient_id):
    print(f"Deleting health information for patient {patient_id}")

@authenticate
def update_patient_health_info(user_type, patient_id, health_info):
    print(f"Updating health information for patient {patient_id}")

# Helper function to check if the user has sufficient permission
def has_sufficient_permission():
    # Placeholder implementation
    return True  # Implement your logic to check for permission

# Placeholder implementation for user authentication
def is_user_authenticated():
    # Replace this with your actual implementation to check user authentication
    return True

# Usage example
if __name__ == '__main__':
    user_type = 'nurse'
    try:
        # Nurse adding patient health information
        add_patient_health_info(user_type, patient_id='123', health_info={})

        # Nurse modifying patient health information
        modify_patient_health_info(user_type, patient_id='123', health_info={})

        # Nurse deleting patient health information
        delete_patient_health_info(user_type, patient_id='123')

        # Nurse updating patient health information
        update_patient_health_info(user_type, patient_id='123', health_info={})
    except Exception as e:
        print(e)
```

In the code above, the `authenticate` decorator checks if the user is authenticated and if the user type is "nurse" before allowing access to the decorated functions. The `authorize` decorator checks if the user type is "nurse" and if the user has sufficient permission to perform the operation.

You can modify the logic inside the decorators (`is_user_authenticated` , `has_sufficient_permission`) according to your application's authentication and authorization mechanism. The user type can be determined based on the user role or any other relevant attribute in the authentication process.

By applying these decorators to the appropriate functions in your EMR application, you can enforce authentication and authorization rules to ensure that only nurses can add, modify, delete, and update patient health information.

We also define an `authorize` decorator that takes a list of allowed roles as an argument. This decorator checks if the user has a role that is included in the allowed roles list before allowing access to the decorated function. If the user has the necessary role, the decorated function is executed; otherwise, an exception is raised.

5.7 Logging and Error Handling Decorators

Logging and error handling decorators are useful for adding logging functionality and handling errors in a centralized manner.

Here's an example of implementing logging and error handling decorators in Python:

```
import logging
from functools import wraps

# Logging decorator
```

```
def log_exception(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        try:
            result = func(*args, **kwargs)
            return result
        except Exception as e:
            logging.error(f"Error occurred in {func.__name__}: {str(e)}")
            raise
    return wrapper

# Error handling decorator
def handle_error(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        try:
            result = func(*args, **kwargs)
            return result
        except Exception as e:
            logging.error(f"Error occurred in {func.__name__}: {str(e)}")
            raise
    return wrapper

# Simulated database
database = {
    "patient123": {
        "fingerprint": "ABCD1234"
    }
}

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(patient_id, fingerprint, *args, **kwargs):
        if verify_fingerprint(patient_id, fingerprint):
            return func(patient_id, fingerprint, *args, **kwargs)
        else:
            raise Exception("Fingerprint verification failed")
    return wrapper

# Verify fingerprint against the database
def verify_fingerprint(patient_id, fingerprint):
    if patient_id in database and database[patient_id]["fingerprint"] == fingerprint:
        return True
    return False

# Example usage
@log_exception
@handle_error
@authenticate
def process_patient_data(patient_id, fingerprint):
    # Process patient data here
    print(f"Processing data for patient: {patient_id}")

# Configure logging
logging.basicConfig(level=logging.ERROR)

# Usage example
if __name__ == '__main__':
    try:
        # Valid patient fingerprint
        process_patient_data("patient123", "ABCD1234")

        # Invalid patient fingerprint
        process_patient_data("patient123", "EFGH5678")
    except Exception as e:
        print("Error occurred:", str(e))
```

In this example, we have the `log_exception` and `handle_error` decorators as mentioned before. We also introduced an `authenticate` decorator to handle the authentication of the patient's fingerprint. The `verify_fingerprint` function checks if the provided fingerprint matches the one in the database.

The `process_patient_data` function is decorated with all three decorators. It receives the `patient_id` and `fingerprint` as arguments. If the fingerprint verification is successful, the function proceeds to process the patient data. If the fingerprint verification fails, an exception is raised.

The example demonstrates two scenarios: one with a valid patient fingerprint and another with an invalid patient fingerprint. In both cases, the decorators log any exceptions that occur.

By using these decorators, you can centralize the logging and error handling logic, making it easier to track and handle exceptions throughout your codebase.

5.8 Decorators for API Rate Limiting and Throttling

API rate limiting and throttling are important techniques used to control the rate of incoming requests to an API in order to prevent abuse or overload. Decorators can be used to implement these functionalities. **Here's an example of implementing decorators for API rate limiting and throttling:**

```
import time
from functools import wraps
```

```
# Rate limiting decorator
def rate_limit(limit, period):
    def decorator(func):
        requests = {}

        @wraps(func)
        def wrapper(patient_id, *args, **kwargs):
            current_time = time.time()

            # Remove expired requests from the dictionary
            for req_time in list(requests.keys()):
                if current_time - float(req_time) > period:
                    del requests[req_time]

            if patient_id in requests and len(requests[patient_id]) >= limit:
                raise Exception("Rate limit exceeded")

            if patient_id not in requests:
                requests[patient_id] = []

            requests[patient_id].append(current_time)

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

        return wrapper

    return decorator

# Example usage
@rate_limit(limit=10, period=60)  # Allow 10 requests per minute
def get_health_info(patient_id):
    # Logic to retrieve health information for the patient
    print(f"Retrieving health information for patient ID: {patient_id}")

# Usage example
if __name__ == '__main__':
    patient_id = '12345'

    try:
        # Make multiple requests to retrieve health information
        for _ in range(15):
            get_health_info(patient_id)
            time.sleep(5)  # Simulate delay between requests

    except Exception as e:
        print(e)
```

In this example, the `rate_limit` decorator is defined to limit the number of requests a patient can make within a specified period. The `limit` parameter determines the maximum number of requests allowed, and the `period` parameter specifies the time window in seconds. If the rate limit is exceeded, an exception is raised.

The `get_health_info` function is decorated with the `rate_limit` decorator, specifying a limit of 10 requests per minute. Each time the function is called with a patient ID, it checks the rate limit and stores the timestamp of the request in a dictionary. If the rate limit is not exceeded, the function proceeds with retrieving the health information.

In the usage example, the `get_health_info` function is called 15 times in quick succession, exceeding the rate limit of 10 requests per minute. As a result, the exception "Rate limit exceeded" is raised.

5.9 Composition and Combination of Decorators

Composition and combination of decorators allow you to apply multiple decorators to a function in a concise and readable way. You can achieve this by using the `@` syntax to stack decorators on top of each other.

Here's an example demonstrating composition and combination of decorators:

```
from functools import wraps

# Decorator 1
def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Decorator 1")
        return func(*args, **kwargs)
    return wrapper

# Decorator 2
def decorator2(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Decorator 2")
        return func(*args, **kwargs)
    return wrapper

# Decorator 3
def decorator3(func):
```

```
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Decorator 3")
        return func(*args, **kwargs)
    return wrapper

# Applying multiple decorators
@decorator1
@decorator2
@decorator3
def my_function():
    print("Inside function")

# Calling the decorated function
my_function()
```

In this example, `my_function` is decorated with three decorators: `decorator1`, `decorator2`, and `decorator3`. The decorators are applied from the bottom up, meaning `decorator3` is applied first, followed by `decorator2`, and finally `decorator1`. When `my_function` is called, it will execute the function along with the behavior added by each decorator in the order they are applied.

The output of the above code will be:

```
Decorator 1
Decorator 2
Decorator 3
Inside function
```

Example:

Here's an example implementation of a component in an EMR application for nurses that incorporates multiple decorators for different functionalities.

Note that this is a simplified example and does not include actual functionality for managing family members, pharmacy orders, etc. It's meant to demonstrate the composition and combination of decorators.

```
import time
from functools import wraps

# Decorator for logging
def log_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Logging: {} function called.".format(func.__name__))
        return func(*args, **kwargs)
    return wrapper

# Decorator for authentication
def authenticate_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Check authentication here
        authenticated = check_authentication()
        if authenticated:
            return func(*args, **kwargs)
        else:
            print("Authentication failed. Access denied.")
    return wrapper

# Decorator for authorization
def authorize_decorator(allowed_roles):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Check authorization here
            authorized = check_authorization(allowed_roles)
            if authorized:
                return func(*args, **kwargs)
            else:
                print("Authorization failed. Access denied.")
        return wrapper
    return decorator

# Decorator for rate limiting
def rate_limit_decorator(limit, interval):
    def decorator(func):
        call_counts = {}

        @wraps(func)
        def wrapper(*args, **kwargs):
            # Check rate limiting here
            key = func.__name__
            if key not in call_counts:
```

```
                call_counts[key] = 0
            call_counts[key] += 1
            if call_counts[key] <= limit:
                return func(*args, **kwargs)
            else:
                print("Rate limit exceeded. Try again later.")

        return wrapper

    return decorator

# Decorator for throttling
def throttle_decorator(interval):
    def decorator(func):
        last_executed = {}

        @wraps(func)
        def wrapper(*args, **kwargs):
            # Check throttling here
            key = func.__name__
            current_time = time.time()
            if key not in last_executed or current_time - last_executed[key] >= interval:
                last_executed[key] = current_time
                return func(*args, **kwargs)
            else:
                print("Request throttled. Try again later.")

        return wrapper

    return decorator

# Placeholder function for checking authentication
def check_authentication():
    # Implement your logic here
    # Return True if authenticated, False otherwise
    return True

# Placeholder function for checking authorization
def check_authorization(allowed_roles):
    # Implement your logic here
    # Check if the user has one of the allowed roles
    # Return True if authorized, False otherwise
    return True

# Functionality for managing family members
@log_decorator
@authenticate_decorator
@rate_limit_decorator(limit=5, interval=60)  # Allow 5 calls per minute
def manage_family_members(patient_id):
    # Logic to manage family members
    print("Managing family members for patient ID:", patient_id)

# Functionality for managing address book
@log_decorator
@authenticate_decorator
@rate_limit_decorator(limit=10, interval=60)  # Allow 10 calls per minute
def manage_address_book(patient_id):
    # Logic to manage address book
    print("Managing address book for patient ID:", patient_id)

# Functionality for managing pharmacy orders
@log_decorator
@authenticate_decorator
@rate_limit_decorator(limit=3, interval=60)  # Allow 3 calls per minute
def manage_pharmacy_orders(patient_id):
    # Logic to manage pharmacy orders
    print("Managing pharmacy orders for patient ID:", patient_id)

# Functionality for managing health documents
@log_decorator
@authenticate_decorator
@rate_limit_decorator(limit=2, interval=60)  # Allow 2 calls per minute
def manage_health_documents(patient_id):
    # Logic to manage health documents
    print("Managing health documents for patient ID:", patient_id)

# Functionality for managing appointments
@log_decorator
@authenticate_decorator
@throttle_decorator(interval=10)  # Allow 1 call per 10 seconds
def manage_appointments(patient_id):
    # Logic to manage appointments
    print("Managing appointments for patient ID:", patient_id)

# Usage example
if __name__ == "__main__":
    patient_id = "12345"

    # Manage family members
    manage_family_members(patient_id)

    # Manage address book
    manage_address_book(patient_id)

    # Manage pharmacy orders
    manage_pharmacy_orders(patient_id)

    # Manage health documents
    manage_health_documents(patient_id)
```

```
    # Manage appointments
    manage_appointments(patient_id)
```

In this example, we have implemented decorators for logging, authentication, rate limiting, and throttling. Each decorator adds specific functionality to the respective functions. The `log_decorator` logs the function calls, `authenticate_decorator` checks for authentication, `rate_limit_decorator` limits the number of calls within a specific time interval, and `throttle_decorator` throttles the requests to a specific rate.

By combining and composing decorators, you can modularize and reuse different aspects of functionality across your codebase, making it more maintainable and flexible.

5.10 Combining Decorators and Context Managers

Combining decorators and context managers allows you to create powerful and flexible patterns in your code. You can use decorators to add functionality to functions or classes, and context managers to manage resources and ensure proper setup and cleanup

Example

Here's an example that demonstrates the combination of decorators and context managers in an EMR application for managing patient health information:

```
from functools import wraps
from contextlib import contextmanager

# Mock functions for demonstration purposes
def is_user_authenticated():
    return True

def has_sufficient_role(allowed_roles):
    return True

def initialize_encryption():
    print("Initializing encryption")

def cleanup_encryption():
    print("Cleaning up encryption")

def start_audit():
    print("Starting audit")

def end_audit():
    print("Ending audit")

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if is_user_authenticated():
            return func(*args, **kwargs)
        else:
            raise Exception("User authentication failed")
    return wrapper

# Authorization decorator
def authorize(allowed_roles):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            if has_sufficient_role(allowed_roles):
                return func(*args, **kwargs)
            else:
                raise Exception("User authorization failed")
        return wrapper
    return decorator

# Encryption context manager
@contextmanager
def encryption_context():
    initialize_encryption()
    try:
        yield
    finally:
        cleanup_encryption()

# Audit context manager
@contextmanager
def audit_context():
    start_audit()
    try:
        yield
    finally:
        end_audit()

# Example usage
@authenticate
def enter_patient_information(patient_id, patient_data):
    with encryption_context(), audit_context():
        # Logic to enter patient information
```

```
        print("Patient information entered:", patient_data)

# Usage example
if __name__ == '__main__':
    patient_id = '12345'
    patient_data = {
        'name': 'John Doe',
        'age': 30,
        'gender': 'Male',
        # Other patient information
    }

    try:
        enter_patient_information(patient_id, patient_data)
    except Exception as e:
        print(e)
```

In this example, we have the `authenticate` decorator to ensure user authentication and the `authorize` decorator to check if the user has sufficient role/permission. We also have two context managers, `encryption_context` and `audit_context`, which provide encryption and audit functionalities, respectively.

The `enter_patient_information` function is decorated with `authenticate` and `authorize` to enforce authentication and authorization. Inside the function, we use the combined context managers `encryption_context` and `audit_context` to apply encryption and auditing to the patient information entry.

By combining decorators and context managers, we can achieve a modular and flexible approach to handle authentication, authorization, encryption, and auditing in our EMR application for managing patient health information.

5.11 Advanced Context Manager Decorator Patterns

Advanced Context Manager Decorator Patterns refer to using context managers in combination with decorators to create more advanced and powerful functionality. Here are a few examples of advanced context manager decorator patterns:

1. **Nested Context Managers:** You can nest multiple context managers within each other to create a hierarchical structure of resource management. This allows you to manage multiple resources in a clean and concise way. Here's an example:

```
@contextmanager
def nested_context_manager():
    with context_manager_1():
        with context_manager_2():
            # Code inside nested context managers
            yield
```

2. **Chained Context Managers:** Instead of nesting context managers, you can chain them together to execute them sequentially. This is useful when you have multiple independent resources that need to be managed. Here's an example:

```
@contextmanager
def chained_context_manager():
    with context_manager_1():
        yield
    with context_manager_2():
        yield
```

3. **Context Managers as Decorators**: You can also use context managers as decorators for functions or methods. This allows you to automatically manage resources before and after the function or method execution. Here's an example:

```
@contextmanager
def context_manager_decorator():
    # Code executed before function/method
    yield
    # Code executed after function/method

@context_manager_decorator
def my_function():
    # Function code
```

4. **Customizing Context Managers:** You can create custom context managers that accept arguments and options to customize their behavior. This allows you to make your context managers more flexible and reusable. Here's an example:

```
@contextmanager
def custom_context_manager(arg1, arg2, option1=False):
    # Code executed before entering the context
    yield
    # Code executed after exiting the context

# Usage example
with custom_context_manager(arg1, arg2, option1=True):
    # Code inside the context
```

Example

```
from functools import wraps
from contextlib import contextmanager

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        if is_user_authenticated():
            return func(*args, **kwargs)
        else:
            raise Exception("User authentication failed")
    return wrapper

# Authorization decorator
def authorize(allowed_roles):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            if has_sufficient_role(allowed_roles):
                return func(*args, **kwargs)
            else:
                raise Exception("User authorization failed")
        return wrapper
    return decorator

# Encryption context manager
@contextmanager
def encryption_context():
    initialize_encryption()
    try:
        yield
    finally:
        cleanup_encryption()

# Audit context manager
@contextmanager
def audit_context():
    start_audit()
    try:
        yield
    finally:
        end_audit()

# Placeholder functions
def initialize_encryption():
    # Implementation to initialize encryption
    pass

def cleanup_encryption():
    # Implementation to cleanup encryption resources
    pass

def start_audit():
    # Implementation to start the audit process
    pass

def end_audit():
    # Implementation to end the audit process
    pass

def is_user_authenticated():
    # Implementation to check if the user is authenticated
    pass

def has_sufficient_role(allowed_roles):
    # Implementation to check if the user has sufficient role/permission
    pass

# Example usage
@authenticate
def enter_patient_information(patient_id, patient_data):
    with encryption_context(), audit_context():
        # Store patient data
        store_patient_data(patient_id, patient_data)
        print("Patient information entered:", patient_data)

@authenticate
def modify_patient_information(patient_id, patient_data):
    with encryption_context(), audit_context():
        # Update patient data
```

```
        update_patient_data(patient_id, patient_data)
        print("Patient information modified:", patient_data)

# Placeholder functions
def store_patient_data(patient_id, patient_data):
    # Placeholder implementation to store patient data
    pass

def update_patient_data(patient_id, patient_data):
    # Placeholder implementation to update patient data
    pass

# Usage example
if __name__ == '__main__':
    patient_id = '12345'
    patient_data = {
        'name': 'John Doe',
        'age': 30,
        'gender': 'Male',
        # Other patient information
    }

    try:
        enter_patient_information(patient_id, patient_data)
        modify_patient_information(patient_id, patient_data)
    except Exception as e:
        print(e)
```

In this example, the `authenticate` decorator ensures that the user is authenticated before accessing the functions `enter_patient_information` and `modify_patient_information`. The `authorize` decorator checks if the user has sufficient role (in this case, 'nurse') to perform the actions.

The `encryption_context` and `audit_context` are used as context managers to handle encryption and audit operations respectively. These context managers automatically handle the setup and cleanup tasks, ensuring that encryption is initialized and auditing is started and ended appropriately.

The EMR application can call these decorated functions to enter and modify patient information, ensuring authentication, authorization, encryption, and audit operations are performed automatically.

Please note that the example provided is a simplified representation, and in a real-world application, the implementation may vary depending on the specific requirements and framework used.

These are just a few examples of the advanced context manager decorator patterns you can use. They provide powerful ways to manage resources, handle exceptions, and control the execution flow of your code.

5.13 Function Wrapping and Inspection

Function wrapping and inspection are important aspects of working with decorators in Python. The `functools` module provides useful tools for wrapping functions and inspecting their properties. Let's explore some of these tools:

1. `wraps`: The `wraps` decorator is used to update the metadata of the wrapper function with that of the original function. It preserves the original function's name, docstring, module, and other attributes.

Here's an example:

```
from functools import wraps

def decorator_function(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Add decorator functionality
        result = func(*args, **kwargs)
        return result
    return wrapper

@decorator_function
def my_function():
    """Original function docstring"""
    # Function body

print(my_function.__name__)  # Output: my_function
print(my_function.__doc__)   # Output: Original function docstring
```

2. `functools.update_wrapper`: The `update_wrapper` function is another way to update the metadata of a wrapper function. It can be used as a standalone function instead of the `wraps` decorator. Here's an example:

```
from functools import update_wrapper

def decorator_function(func):
    def wrapper(*args, **kwargs):
        # Add decorator functionality
        result = func(*args, **kwargs)
        return result

    # Update the wrapper function's metadata with that of the original function
    update_wrapper(wrapper, func)

    return wrapper

@decorator_function
def my_function():
    """Original function docstring"""
    # Function body

print(my_function.__name__)  # Output: my_function
print(my_function.__doc__)   # Output: Original function docstring
```

3. `inspect` : The `inspect` module provides functions for inspecting the properties of functions and other objects. It can be used to retrieve information such as the arguments, return type, and source code of a function.

Here's an example:

```
import inspect

def my_function(arg1, arg2):
    """Original function"""
    return arg1 + arg2

# Get the function signature
signature = inspect.signature(my_function)
print(str(signature))  # Output: (arg1, arg2)

# Get the function's docstring
docstring = inspect.getdoc(my_function)
print(docstring)  # Output: Original function

# Get the source code of the function
source_code = inspect.getsource(my_function)
print(source_code)
```

The `inspect` module provides various other functions, such as `getargspec` , `getfullargspec` , `getsourcefile` , etc., which allow you to inspect different aspects of a function.

These tools enable you to wrap functions while preserving their original metadata and inspect their properties for introspection or debugging purposes.

5.14 Manipulating Function Execution Flow

Manipulating function execution flow in Python can be achieved using various techniques. Here are a few examples:

1. **Decorators**: Decorators allow you to modify the behavior of a function by wrapping it with additional functionality. You can use decorators to execute code before and/or after the wrapped function, modify input arguments, or handle exceptions. Decorators provide a way to manipulate the function execution flow without modifying the original function's code.

2. **Context Managers:** Context managers provide a way to define setup and teardown actions around a block of code. By implementing the `__enter__` and `__exit__` methods, you can control the execution flow before and after the block of code. Context managers are commonly used with the `with` statement to ensure resources are properly managed.

3. **Conditional Statements:** Using conditional statements such as `if`, `elif`, and `else`, you can control the execution flow based on certain conditions. By evaluating conditions, you can decide which blocks of code should be executed.

4. **Exception Handling**: With `try`, `except`, `else`, and `finally` statements, you can handle exceptions and control the execution flow based on whether an exception occurs or not. Exception handling allows you to define alternate paths of execution when specific errors occur.

5. **Function Composition**: Function composition is the act of combining multiple functions to create a new function. By composing functions together, you can control the flow of data and pass outputs of one function as inputs to another. Function composition enables you to create complex behavior by combining simpler functions.

Example

```
from functools import wraps

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(user_type, *args, **kwargs):
        if user_type == 'nurse' and is_user_authenticated():
            return func(user_type, *args, **kwargs)
        else:
            raise Exception("User authentication failed")
    return wrapper

# Authorization decorator
def authorize(allowed_roles):
    def decorator(func):
        @wraps(func)
        def wrapper(user_type, *args, **kwargs):
            if user_type == 'nurse' and has_sufficient_permission():
                return func(user_type, *args, **kwargs)
            else:
                raise Exception("User authorization failed")
        return wrapper
    return decorator

# Example usage
@authenticate
def add_patient_health_info(user_type, patient_id, health_info):
    # Logic to add patient health information
    print("Adding patient health info:", patient_id, health_info)

@authenticate
def modify_patient_health_info(user_type, patient_id, health_info):
    # Logic to modify patient health information
    print("Modifying patient health info:", patient_id, health_info)

# Helper function to check if the user has sufficient role/permission
def has_sufficient_permission():
    # Placeholder implementation
    return True

# Usage example
if __name__ == '__main__':
    user_type = 'nurse'
    try:
        # Nurse adding patient health information
        add_patient_health_info(user_type, patient_id='123', health_info={})

        # Nurse modifying patient health information
        modify_patient_health_info(user_type, patient_id='123', health_info={})
    except Exception as e:
        print(e)
```

In this example, we have defined two decorators: `authenticate` and `authorize`. These decorators are used to enforce user authentication and authorization before allowing access to certain functions. The decorators check the user's `user_type` and call the respective helper functions (`is_user_authenticated` and `has_sufficient_permission`) to perform the necessary checks.

The functions `add_patient_health_info` and `modify_patient_health_info` are decorated with `@authenticate`, ensuring that only authenticated nurses can access these functions. Additionally, the `has_sufficient_permission` function is used to check if the nurse has the required permission to perform the actions.

The usage example demonstrates how the decorators can be applied to the functions and how the authentication and authorization checks are performed before executing the function logic. If authentication or authorization fails, an exception is raised.

These techniques provide flexibility in manipulating the execution flow of functions in Python and allow you to customize the behavior according to your requirements.

5.15 Decorator Optimizations and Performance Tuning

When working with decorators, it's important to consider their performance implications. Here are some optimizations and performance tuning techniques for decorators:

1. **Avoid unnecessary function calls**: Minimize the number of function calls within decorators. Each additional function call adds overhead, so try to keep the decorator implementation concise.

2. **Use functools.wraps:** The `wraps` decorator from the `functools` module preserves the original function's metadata, such as the name, docstring, and arguments. It helps maintain consistency and improves debugging. Apply `@wraps(func)` to the wrapper function in your decorator.

3. **Decorator factory for customization:** If your decorator needs additional configuration or customization, you can create a decorator factory. A decorator factory is a function that returns a decorator based on the provided arguments. It allows flexibility in configuring the decorator behavior.

4. **Decorator chaining:** You can chain multiple decorators by applying multiple decorators to a function. Decorators are applied from bottom to top, so the innermost decorator is executed first. This allows you to combine multiple functionalities in a modular way.

5. **Caching:** If your decorator involves computationally expensive operations or requires frequent lookups, consider implementing caching mechanisms to store and reuse results. This can significantly improve performance by avoiding redundant computations.

6. **Lazy evaluation:** Delay the evaluation of heavy computations or resource-intensive operations until they are actually needed. Use lazy evaluation techniques such as generators or lazy loading to defer the execution until necessary.

7. **Use optimized data structures:** Choose appropriate data structures for efficient data manipulation and retrieval. Consider using data structures like sets, dictionaries, or data structures from specialized libraries for better performance in specific use cases.

8. **Profile and optimize:** Profile your decorator code using profiling tools to identify performance bottlenecks. Optimize the critical sections of your code to improve performance. Techniques like algorithmic optimizations, data structure optimizations, or using low-level constructs can provide significant performance gains.

Remember, optimizing decorators should be done based on the specific requirements and performance constraints of your application. It's recommended to measure the performance impact of your decorators and profile the code to identify areas for improvement.

```
import functools
import time

# Decorator with functools.wraps
def my_decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # Perform some decoration
        print("Executing decorator")
        return func(*args, **kwargs)
    return wrapper

# Decorator factory for customization
def my_decorator_factory(option):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            if option == 'A':
                print("Option A")
            elif option == 'B':
                print("Option B")
            return func(*args, **kwargs)
        return wrapper
    return decorator

# Chaining decorators
@my_decorator
@my_decorator_factory(option='A')
def my_function():
    print("Inside function")

# Caching decorator
def cache_decorator(func):
    cache = {}

    @functools.wraps(func)
    def wrapper(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result

    return wrapper

@cache_decorator
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Lazy evaluation decorator
def lazy_evaluation_decorator(func):
    result = None

    @functools.wraps(func)
    def wrapper():
        nonlocal result
        if result is None:
            result = func()
        return result

    return wrapper

@lazy_evaluation_decorator
def heavy_computation():
    time.sleep(5)  # Simulate a heavy computation
    return 42

# Usage example
if __name__ == '__main__':
    # Decorator execution
    my_function()
```

```
    # Decorator factory
    decorated_func = my_decorator_factory(option='B')(my_function)
    decorated_func()

    # Caching decorator
    print(fibonacci(10))
    print(fibonacci(10))  # Uses cached result

    # Lazy evaluation decorator
    print(heavy_computation())  # Executes heavy computation
    print(heavy_computation())  # Returns cached result
```

In this example, we have demonstrated the usage of `functools.wraps` to preserve function metadata, a decorator factory for customization, decorator chaining, caching with a decorator, and lazy evaluation with a decorator.

Example

```
import time
from functools import lru_cache, wraps

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(user_type, *args, **kwargs):
        if user_type == 'nurse' and is_user_authenticated():
            return func(user_type, *args, **kwargs)
        else:
            raise Exception("User authentication failed")
    return wrapper

# Authorization decorator
def authorize(func):
    @wraps(func)
    def wrapper(user_type, *args, **kwargs):
        if user_type == 'nurse' and has_sufficient_permission():
            return func(user_type, *args, **kwargs)
        else:
            raise Exception("User authorization failed")
    return wrapper

# Decorator for logging function calls
def log_function_call(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

# Decorator for measuring execution time
def measure_execution_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Function {func.__name__} executed in {execution_time} seconds")
        return result
    return wrapper

# Example usage
@authenticate
@log_function_call
@measure_execution_time
@lru_cache(maxsize=128)
def retrieve_patient_info(user_type, patient_id):
    # Simulate fetching patient information from the database
    time.sleep(1)
    return get_patient_info_from_database(patient_id)

# Helper function to check if the user has sufficient permission
def has_sufficient_permission():
    # Placeholder implementation
    user_role = get_user_role()  # Implement your logic to get the user role
    allowed_roles = ['nurse']
    return user_role in allowed_roles

# Helper function to check if the user is authenticated
def is_user_authenticated():
    # Placeholder implementation
    # Implement your authentication logic here
    # For example, check if the user is logged in or has valid credentials
    return True  # Return True if authenticated, False otherwise

# Helper function to get the user role
def get_user_role():
    # Placeholder implementation
    # Implement your logic to get the user role
    return 'nurse'  # Return the user role
```

```
# Helper function to get patient information from the database
def get_patient_info_from_database(patient_id):
    # Placeholder implementation
    # Implement your logic to fetch patient information from the database
    return {'patient_id': patient_id, 'name': 'John Doe', 'age': 35, 'gender': 'Male'}

# Usage example
if __name__ == '__main__':
    patient_id = '12345'

    try:
        info = retrieve_patient_info('nurse', patient_id)
        print("Patient info:", info)
    except Exception as e:
        print("An error occurred:", str(e))
```

In this example, we have applied the decorator optimizations and performance tuning techniques to an EMR application for nurses. We have used the `lru_cache` decorator from `functools` to cache the results of the `retrieve_patient_info` function, reducing redundant database queries. We have also applied the authentication decorator, logging decorator, and execution time measurement decorator to enhance the functionality of the function.

These optimizations and techniques can help improve the performance and efficiency of your decorators and overall application in real-world scenarios.

5.16 Overcoming Limitations and Extending Decorator Functionality

Here's an example of how to overcome limitations and extend the functionality of decorators in an EMR application for nurses:

```
from functools import wraps

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(user_type, *args, **kwargs):
        if user_type == 'nurse' and is_user_authenticated():
            return func(user_type, *args, **kwargs)
        else:
            raise Exception("User authentication failed")
    return wrapper

# Authorization decorator
def authorize(allowed_roles):
    def decorator(func):
        @wraps(func)
        def wrapper(user_type, *args, **kwargs):
            if user_type == 'nurse' and has_sufficient_permission(allowed_roles):
                return func(user_type, *args, **kwargs)
            else:
                raise Exception("User authorization failed")
        return wrapper
    return decorator

# Decorator for logging function calls
def log_function_call(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

# Decorator for caching function results
def cache_result(func):
    @wraps(func)
    def wrapper(user_type, *args, **kwargs):
        # Generate a cache key based on the function name and arguments
        cache_key = f"{func.__name__}:{args}:{kwargs}"

        # Check if the result is already cached
        if cache_key in cache:
            return cache[cache_key]

        # Execute the function and cache the result
        result = func(user_type, *args, **kwargs)
        cache[cache_key] = result

        return result
    return wrapper

# Example usage
@authenticate
@log_function_call
@cache_result
def retrieve_patient_info(user_type, patient_id):
    # Simulate fetching patient information from the database
    return get_patient_info_from_database(patient_id)

# Helper function to check if the user has sufficient permission
def has_sufficient_permission(allowed_roles):
    # Placeholder implementation
    user_role = get_user_role()  # Implement your logic to get the user role
```

```
    return user_role in allowed_roles

# Helper function to check if the user is authenticated
def is_user_authenticated():
    # Placeholder implementation
    # Implement your authentication logic here
    # For example, check if the user is logged in or has valid credentials
    return True  # Return True if authenticated, False otherwise

# Helper function to get the user role
def get_user_role():
    # Placeholder implementation
    # Implement your logic to get the user role
    return 'nurse'  # Return the user role

# Helper function to get patient information from the database
def get_patient_info_from_database(patient_id):
    # Placeholder implementation
    # Implement your logic to fetch patient information from the database
    return {'patient_id': patient_id, 'name': 'John Doe', 'age': 35, 'gender': 'Male'}

# Cache to store function results
cache = {}

# Usage example
if __name__ == '__main__':
    patient_id = '12345'

    try:
        info = retrieve_patient_info('nurse', patient_id)
        print("Patient info:", info)

        # Calling the function again should retrieve the cached result
        info_cached = retrieve_patient_info('nurse', patient_id)
        print("Cached patient info:", info_cached)
    except Exception as e:
        print("An error occurred:", str(e))
```

In this example, I have extended the decorator functionality by adding a `cache_result` decorator that caches the results of the function in a dictionary called `cache`. This can help improve performance by avoiding redundant function executions for the same input arguments.

Additionally, the `allowed_roles` parameter is passed to the `authorize` decorator, which allows you to specify the roles that are authorized to access the decorated function.

Remember to replace the placeholder implementations (`has_sufficient_permission`, `is_user_authenticated`, and `get_patient_info_from_database`) with your actual implementations based on your EMR application's requirements.

5.17 Handling Decorator State and Persistent Data

Handling decorator state and persistent data can be achieved by utilizing various techniques. Here's an example of how you can implement it in the context of an EMR application for nurses:

```
from functools import wraps

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(user_type, *args, **kwargs):
        if user_type == 'nurse' and is_user_authenticated():
            return func(user_type, *args, **kwargs)
        else:
            raise Exception("User authentication failed")
    return wrapper

# Authorization decorator
def authorize(allowed_roles):
    def decorator(func):
        @wraps(func)
        def wrapper(user_type, *args, **kwargs):
            if user_type == 'nurse' and has_sufficient_permission(allowed_roles):
                return func(user_type, *args, **kwargs)
            else:
                raise Exception("User authorization failed")
        return wrapper
    return decorator

# Decorator for logging function calls
def log_function_call(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Calling function: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

# Decorator for storing persistent data
def store_persistent_data(func):
    @wraps(func)
```

```
    def wrapper(user_type, *args, **kwargs):
        # Store the persistent data in a dictionary specific to the user
        user_data = get_user_data(user_type)
        user_data['persistent_data'] = "Some persistent data"
        # Pass the persistent data to the decorated function
        result = func(user_type, user_data, *args, **kwargs)
        return result
    return wrapper

# Example usage
@authenticate
@log_function_call
@store_persistent_data
def enter_patient_data(user_type, user_data, patient_id, data):
    # Access the persistent data
    persistent_data = user_data.get('persistent_data')
    print("Persistent data:", persistent_data)
    # Logic to enter patient data
    print(f"Entering patient data for patient {patient_id}: {data}")

# Helper function to check if the user has sufficient permission
def has_sufficient_permission(allowed_roles):
    # Placeholder implementation
    user_role = get_user_role()  # Implement your logic to get the user role
    return user_role in allowed_roles

# Helper function to check if the user is authenticated
def is_user_authenticated():
    # Placeholder implementation
    # Implement your authentication logic here
    # For example, check if the user is logged in or has valid credentials
    return True  # Return True if authenticated, False otherwise

# Helper function to get the user role
def get_user_role():
    # Placeholder implementation
    # Implement your logic to get the user role
    return 'nurse'  # Return the user role

# Helper function to get user-specific data
def get_user_data(user_type):
    # Placeholder implementation
    # Retrieve user-specific data from a storage mechanism
    # For example, retrieve data based on the user type
    if user_type == 'nurse':
        return {'username': 'nurse123', 'persistent_data': None}
    else:
        return {}

# Usage example
if __name__ == '__main__':
    user_type = 'nurse'
    patient_id = '12345'
    data = {'name': 'John Doe', 'age': 35, 'gender': 'Male'}

    try:
        enter_patient_data(user_type, patient_id, data)
    except Exception as e:
        print("An error occurred:", str(e))
```

In this example, the `store_persistent_data` decorator is used to store persistent data for each user in a dictionary. The `user_data` dictionary is passed as an argument to the decorated function, allowing access to the persistent data within the function. You can customize the `get_user_data` function to retrieve user-specific data based on the `user_type` parameter.

6. Advanced Caching Techniques and Strategies

Advanced caching techniques and strategies can further optimize the caching process and improve performance in specific scenarios. Here are some advanced techniques and strategies for caching:

1. **LRU Cache (Least Recently Used Cache):**
— LRU cache is a popular caching strategy that keeps track of the most recently used items.
— When the cache reaches its capacity, the least recently used item is evicted to make room for new items.
— Python provides an `lru_cache` decorator from the `functools` module that implements the LRU cache strategy.

2. **Time-Based Expiration:**
— In some cases, cached data may have a limited validity period and needs to be refreshed after a certain time.
— You can implement time-based expiration by associating an expiration timestamp with each cached item.
— Before accessing a cached item, check its expiration timestamp and refresh the item if it has expired.

3. **Cache Invalidation:**
— Cache invalidation is the process of removing or updating cached data when it becomes stale or outdated.
— You can implement cache invalidation by associating a version number or a timestamp with each cached item.
— When the underlying data changes, update the version number or timestamp, and invalidate the corresponding cached items.

4. **Partial Caching:**
— Instead of caching the entire result of a function, you can selectively cache only parts of the result that are expensive to compute.
— Break down the function into smaller components or sub-results, and cache only those components that provide significant performance gains.

5. **Distributed Caching:**
— In distributed systems, caching can be distributed across multiple servers or nodes to improve scalability and reduce load on individual servers.
— Distributed caching solutions like Redis, Memcached, or distributed cache frameworks can be used to manage caching across multiple nodes.

6. **Cache Preloading:**
— In scenarios where the application startup time is critical, you can preload the cache with frequently accessed data during the application startup phase.
— This ensures that the cache is warm and ready to serve requests immediately, avoiding initial cache misses.

7. **Cache Partitioning:**
— If the cache size becomes too large or the access pattern is unevenly distributed, you can partition the cache into multiple smaller caches.
— Each partition can be assigned to different segments of data, allowing for better cache management and reducing contention.

Example

```
from functools import lru_cache
import time

# Placeholder functions for retrieving patient demographics and lab test results
def get_patient_demographics(patient_id):
    # Placeholder implementation to fetch patient demographics from the database
    print(f"Fetching patient demographics for ID: {patient_id}")
    return patient_demographics

def get_lab_test_results(patient_id):
    # Placeholder implementation to fetch lab test results from the database
    print(f"Fetching lab test results for ID: {patient_id}")
    return lab_test_results

# Placeholder patient demographics and lab test results
patient_demographics = {
    'patient_id': '12345',
    'name': 'John Doe',
    'age': 30,
    'gender': 'Male',
    # Other patient demographic information
}

lab_test_results = {
    'patient_id': '12345',
    'results': [
        {'test_type': 'Blood Test', 'result': 'Normal'},
        {'test_type': 'X-ray', 'result': 'Abnormal'},
        # Other lab test results
    ]
}

# LRU Cache for patient demographic information
@lru_cache(maxsize=128)
def retrieve_patient_demographics(patient_id):
    # Simulated expensive operation to fetch patient demographics
    time.sleep(1)
    return get_patient_demographics(patient_id)

# Time-based expiration for lab test results cache
lab_results_cache = {}
CACHE_EXPIRATION_TIME = 3600  # 1 hour

def retrieve_lab_test_results(patient_id):
    if patient_id in lab_results_cache:
        result, timestamp = lab_results_cache[patient_id]
        if time.time() - timestamp < CACHE_EXPIRATION_TIME:
            print(f"Fetching lab test results from cache for ID: {patient_id}")
            return result

    # Simulated expensive calculation
    time.sleep(1)
    result = get_lab_test_results(patient_id)

    # Update cache with new result and timestamp
    lab_results_cache[patient_id] = (result, time.time())
    return result

# Cache invalidation when updating patient health information
def update_patient_health_info(patient_id, health_info):
    # Placeholder implementation to update patient health information in the database
    print(f"Updating patient health information for ID: {patient_id}")
    # Perform the necessary update operations in the database

    # Invalidate cached patient demographics for the updated patient
    retrieve_patient_demographics.cache_clear()
    # Invalidate cached lab test results for the updated patient
    lab_results_cache.pop(patient_id, None)
```

```
    print(f"Patient health information updated for ID: {patient_id}")

# Usage example
if __name__ == '__main__':
    patient_id = '12345'

    # Nurse accessing patient demographics
    demographics = retrieve_patient_demographics(patient_id)
    print("Patient demographics:", demographics)

    # Nurse accessing lab test results
    lab_results = retrieve_lab_test_results(patient_id)
    print("Lab test results:", lab_results)

    # Nurse updating patient health information
    update_patient_health_info(patient_id, health_info={})

    # Nurse accessing patient demographics after update (cache cleared)
    demographics = retrieve_patient_demographics(patient_id)
    print("Updated patient demographics:", demographics)

    # Nurse accessing lab test results after update (cache invalidated)
    lab_results = retrieve_lab_test_results(patient_id)
    print("Updated lab test results:", lab_results)
```

These techniques and strategies can be combined and tailored to specific use cases to achieve optimal caching performance. It's important to analyze the requirements, data access patterns, and scalability needs of your application to choose the most appropriate caching approach.

6.3 Dynamic Debugging and Profiling with Decorators

Dynamic debugging and profiling with decorators can be achieved using libraries like `pyinstrument` and `line_profiler`. Here's an example that demonstrates the usage of `pyinstrument` for profiling and timing function execution:

```
from functools import wraps
import cProfile

# Profiling Decorator
def profile(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        profiler = cProfile.Profile()
        profiler.enable()
        result = func(*args, **kwargs)
        profiler.disable()
        profiler.print_stats()
        return result
    return wrapper

# Timing Decorator
def measure_execution_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Function {func.__name__} executed in {execution_time} seconds")
        return result
    return wrapper

# Example usage
@profile
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

if __name__ == '__main__':
    result = fibonacci(10)
    print("Result:", result)
```

In this example, the `profile` decorator uses `pyinstrument` to profile the function execution and print the profiling results. The `measure_execution_time` decorator measures the execution time of the function and prints it.

You can customize the decorators and integrate them into your EMR application for nurses to dynamically debug and profile specific functions or sections of code.

Note that you'll need to install `pyinstrument` library using `pip` before running the code: `pip install pyinstrument`.

Please note that dynamic debugging and profiling can have performance overhead, so it's recommended to use them selectively and for specific debugging or optimization purposes.

6.4 Decorator-Driven Code Instrumentation

Decorator-driven code instrumentation refers to the process of modifying the behavior of functions or methods using decorators to add instrumentation code. This allows you to gather additional information or perform certain actions during the execution of the decorated code.

Here's an example of decorator-driven code instrumentation using a decorator to measure the execution time of a function:

```
import time
from functools import wraps

# Timing Decorator
def measure_execution_time(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Function {func.__name__} executed in {execution_time} seconds")
        return result
    return wrapper

# Example usage
@measure_execution_time
def my_function():
    time.sleep(2)
    print("Function execution")

if __name__ == '__main__':
    my_function()
```

In this example, the `measure_execution_time` decorator is defined to wrap a function and measure its execution time. The decorator creates a wrapper function that calculates the execution time before and after calling the original function. It then prints the execution time and returns the result.

By applying the `@measure_execution_time` decorator to the `my_function` function, the execution time of `my_function` will be measured and printed when it is called.

You can extend this concept of decorator-driven code instrumentation to add additional functionality or gather specific metrics based on your requirements. For example, you could create decorators to log function calls, count the number of times a function is executed, track input/output values, or perform any other custom actions.

Decorators provide a flexible and non-intrusive way to modify the behavior of functions or methods, making them a powerful tool for code instrumentation and adding functionality to your codebase.

6.5 Advanced Tracing and Logging Techniques

Advanced tracing and logging techniques can be invaluable for debugging and analyzing the behavior of complex systems.

Here's an example of how you can implement tracing and logging using decorators:

```
import logging
from functools import wraps

# Configure logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s [%(levelname)s] %(message)s')

# Tracing Decorator
def trace(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        logging.debug(f"Calling function: {func.__name__}")
        result = func(*args, **kwargs)
        logging.debug(f"Function {func.__name__} completed")
        return result
    return wrapper

# Example usage
@trace
def calculate_sum(a, b):
    return a + b

@trace
def calculate_product(a, b):
    return a * b

if __name__ == '__main__':
    logging.info("Starting program")

    result = calculate_sum(3, 4)
    logging.info(f"Sum: {result}")
```

```
    result = calculate_product(3, 4)
    logging.info(f"Product: {result}")

    logging.info("Program completed")
```

In this example, we define a `trace` decorator that logs the start and completion of a function using the `logging` module. The `trace` decorator is then applied to the `calculate_sum` and `calculate_product` functions.

When the program is run, it will produce log messages indicating the function calls and their results. The log level is set to `DEBUG`, but you can adjust it according to your needs. The log format can also be customized to include additional information if desired.

Using advanced tracing and logging techniques like this can help you gain insights into the flow and behavior of your application, making it easier to identify and debug issues.

6.6 Integrating Decorators with Profiling Frameworks

Integrating decorators with profiling frameworks allows you to easily apply profiling to specific functions or methods in your codebase.

Here's an example of integrating decorators with the `cProfile` module, a built-in profiling tool in Python:

```
import cProfile
import pstats
from functools import wraps

# Profiling Decorator
def profile(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        profiler = cProfile.Profile()
        profiler.enable()
        result = func(*args, **kwargs)
        profiler.disable()
        stats = pstats.Stats(profiler)
        stats.print_stats()
        return result
    return wrapper

# Example usage
@profile
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

if __name__ == '__main__':
    fibonacci(10)
```

In this example, we define a `profile` decorator that uses the `cProfile` module to profile the execution of a function. The decorator enables the profiler before calling the function and disables it afterward. The profiling results are then printed using the `pstats.Stats` class.

By applying the `profile` decorator to the `fibonacci` function, we can profile its execution and see the profiling statistics, such as the number of function calls, total time spent, and time per function call.

You can customize the decorator based on your profiling requirements and use other profiling frameworks such as `line_profiler` or `pyinstrument` by following a similar approach.

Integrating decorators with profiling frameworks gives you the flexibility to selectively profile specific functions or methods in your codebase, helping you identify performance bottlenecks and optimize your application.

7 Profiling and Performance Analysis with Decorators

Profiling and performance analysis are crucial for optimizing the performance of applications, including EMR systems for nurses. Decorators can be used to integrate profiling and performance analysis tools into the codebase. Let's see an example:

```
import time
import cProfile

# Profiling decorator
def profile(func):
    def wrapper(*args, **kwargs):
        profiler = cProfile.Profile()
```

```
        profiler.enable()

        result = func(*args, **kwargs)

        profiler.disable()
        profiler.print_stats()

        return result

    return wrapper

# Example usage
@profile
def process_patient_data(patient_data):
    # Simulate processing patient data
    time.sleep(1)
    # Perform time-consuming operations
    # ...

    return "Processing complete"

# Usage example
if __name__ == '__main__':
    patient_data = {}  # Placeholder patient data

    result = process_patient_data(patient_data)
    print(result)
```

In this example, we have a decorator `@profile` applied to the `process_patient_data` function. The `@profile` decorator uses the `cProfile` module from the Python standard library to perform profiling of the decorated function.

When the `process_patient_data` function is called, the `@profile` decorator wraps the function execution with profiling functionality. It creates a `cProfile.Profile` object, enables profiling, executes the function, disables profiling, and then prints the profiling statistics using `profiler.print_stats()`.

Profiling allows you to gather detailed performance data, such as the number of function calls, execution times, and memory usage. By analyzing the profiling results, you can identify bottlenecks and optimize the performance of critical sections of code.

To run the profiling-enabled code, you can execute the script as usual. The profiling results will be printed to the console after the function execution is complete.

Note: Profiling can add overhead to the execution of your code, so it is recommended to use it selectively on specific functions or sections of code where you suspect performance issues. It's also important to interpret and analyze the profiling results carefully to identify the areas that need optimization.

In an EMR application for nurses, profiling can help identify performance bottlenecks in critical operations, such as data processing, database access, or computationally intensive tasks. By using decorators for profiling, you can easily integrate performance analysis into your codebase and gain insights to improve the overall performance of the application.

7.2 Decorator-Based Performance Optimizations

Here's an example of how decorators can be used for performance optimizations in a real-world EMR application for nurses:

```
import time
from functools import wraps

# Performance optimization decorator
def optimize_performance(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()

        # Execute the function
        result = func(*args, **kwargs)

        end_time = time.time()
        execution_time = end_time - start_time

        # Log the execution time
        print(f"Function {func.__name__} executed in {execution_time} seconds")

        return result

    return wrapper

# Example usage

# Fake data retrieval function
def retrieve_patient_data(patient_id):
    # Simulate fetching patient data from the database
    time.sleep(1)
    return {"patient_id": patient_id, "name": "John Doe", "age": 35, "gender": "Male"}

# Function to process patient data
@optimize_performance
```

```
def process_patient_data(patient_id):
    patient_data = retrieve_patient_data(patient_id)

    # Perform data processing tasks
    # ...

    # Simulate some time-consuming operations
    time.sleep(2)

    # Return the processed data
    return {"processed_data": "Some processed data"}

# Usage example
if __name__ == "__main__":
    patient_id = "12345"

    # Process the patient data
    processed_data = process_patient_data(patient_id)
    print("Processed data:", processed_data)
```

In this example, we have a decorator `@optimize_performance` that measures and logs the execution time of the decorated function. The decorator wraps the function execution, records the start and end times, calculates the execution time, and logs it to the console.

The `process_patient_data` function is decorated with `@optimize_performance`, which means that the execution time of this function will be logged when it is called. The function retrieves patient data, performs data processing tasks, simulates time-consuming operations, and returns the processed data.

By applying the `@optimize_performance` decorator to the `process_patient_data` function, we can easily track the execution time and identify any performance bottlenecks. This information can help in optimizing the code to improve the overall performance of the EMR application.

By selectively applying the `@optimize_performance` decorator to specific functions or sections of code that are critical for performance, you can gain insights into the execution time and identify areas for improvement. This approach allows you to focus on optimizing the most impactful parts of the application while keeping the codebase clean and maintainable.

7.3 Memory Management and Resource Optimization

Memory Management and Resource Optimization are crucial aspects of application development to ensure efficient utilization of system resources and prevent resource leaks. Decorators can be utilized to implement various techniques for memory management and resource optimization. Here's an example of a real-world application in EMR for nurses that demonstrates memory management using a decorator:

```
import psutil
import random

# Memory usage decorator
def measure_memory_usage(func):
    def wrapper(*args, **kwargs):
        # Get the initial memory usage
        start_memory = psutil.Process().memory_info().rss / 1024  # in KB

        # Call the function
        result = func(*args, **kwargs)

        # Get the final memory usage
        end_memory = psutil.Process().memory_info().rss / 1024  # in KB

        # Calculate the memory usage difference
        memory_usage = end_memory - start_memory

        # Print the memory usage
        print(f"Memory usage: {memory_usage} KB")

        return result

    return wrapper

# Example usage
@measure_memory_usage
def process_patient_data(patient_data):
    # Perform some processing on patient data
    processed_data = perform_processing(patient_data)
    return processed_data

# Placeholder function for processing patient data
def perform_processing(patient_data):
    # Placeholder implementation
    # Perform actual processing on the patient data
    processed_data = patient_data  # Placeholder result
    return processed_data

# Placeholder function for generating patient data
def generate_patient_data():
    # Placeholder implementation
    # Generate and return fake patient data
    patient_data = {
        'name': 'John Doe',
```

```
        'age': 35,
        'gender': 'Male',
        # Additional patient information
    }
    return patient_data

# Usage example
if __name__ == '__main__':
    # Generate some fake patient data
    patient_data = generate_patient_data()

    # Process the patient data
    processed_data = process_patient_data(patient_data)

    # Perform other operations with the processed data
    # ...
```

In this example, the `measure_memory_usage` decorator measures the memory usage of the `process_patient_data` function. It uses the `resource` module to obtain the maximum resident set size (memory usage) of the current process before and after the function execution. The difference between the two measurements represents the memory usage of the function.

By applying the `measure_memory_usage` decorator to the `process_patient_data` function, we can easily track and monitor the memory consumption of that function. This can help identify any memory-intensive operations and optimize the code accordingly.

Remember to adapt the `perform_processing` and `generate_patient_data` functions according to your specific requirements in the EMR application.

7.4 Parallel Execution and Distributed Computing with Decorators

Parallel execution and distributed computing are techniques used to enhance the performance and scalability of applications by leveraging multiple computing resources. Decorators can be used to simplify the implementation of parallel execution and distributed computing patterns.

To demonstrate the concept, let's consider an example of processing patient data in parallel using the `concurrent.futures` module, which provides a high-level interface for asynchronously executing callables.

Here's an example code that showcases parallel execution using decorators:

```
import concurrent.futures

def process_data(data):
    # Perform some time-consuming task
    result = perform_processing(data)
    return result

@parallel_execution
def process_data_in_parallel(data_list):
    with concurrent.futures.ProcessPoolExecutor() as executor:
        results = executor.map(process_data, data_list)
    return list(results)

@distributed_computing
def process_data_distributed(data_list):
    with concurrent.futures.ThreadPoolExecutor() as executor:
        results = executor.map(process_data, data_list)
    return list(results)

# Example usage
data_list = generate_data_list()
parallel_results = process_data_in_parallel(data_list)
distributed_results = process_data_distributed(data_list)
```

In this example, we have a `process_data` function that performs a time-consuming task on a single data item. We then define decorators `parallel_execution` and `distributed_computing` to apply parallel execution and distributed computing techniques, respectively, to a function that processes a list of data.

The `process_data_in_parallel` function uses the `concurrent.futures.ProcessPoolExecutor` to parallelize the execution of `process_data` function across multiple processes.

The `process_data_distributed` function uses the `concurrent.futures.ThreadPoolExecutor` to distribute the execution of `process_data` function across multiple threads.

By applying the decorators `parallel_execution` and `distributed_computing` to these functions, we can easily leverage parallelism and distributed computing capabilities to process large datasets or perform computationally intensive tasks efficiently.

These techniques can be applied in various real-world applications, such as data processing pipelines, machine learning, scientific computing, and more, where parallel execution and distributed computing are crucial for improving performance and scalability.

8 Decorators in Frameworks and Libraries

Decorators play a crucial role in many frameworks and libraries, enabling developers to extend and customize their functionality.

1. Flask (Web Framework): Flask uses decorators extensively for defining routes. The `@app.route` decorator is used to bind a URL pattern to a function, making it a route handler. Decorators are also used for middleware, authentication, caching, and error handling.

Here are a few examples of how decorators are used in popular frameworks and libraries:

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run()
```

2. **Django (Web Framework):** Django uses decorators for various purposes, including authentication, permission checks, caching, and view registration. Decorators like `@login_required` and `@permission_required` ensure that only authenticated users with the necessary permissions can access certain views.

```
from django.contrib.auth.decorators import login_required, permission_required
from django.shortcuts import render

@login_required
@permission_required('myapp.can_view_data')
def view_data(request):
    data = fetch_data_from_database()
    return render(request, 'data.html', {'data': data})
```

3. **SQLAlchemy (Database Toolkit):** SQLAlchemy uses decorators for mapping Python classes to database tables. The `@sqlalchemy.orm.mapper` decorator is used to define the mapping between the class attributes and the database columns.

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    username = Column(String)
    password = Column(String)

    def __repr__(self):
        return f"<User(id={self.id}, username={self.username})>"
```

4. **Celery (Distributed Task Queue):** Celery uses decorators for defining and configuring asynchronous tasks. The `@celery.task` decorator marks a function as a Celery task, allowing it to be executed asynchronously and distributed across a worker pool.

```
from celery import Celery

app = Celery('myapp', broker='redis://localhost:6379/0')

# Placeholder function for processing data
def perform_processing(data):
    # Placeholder implementation of time-consuming task
    # Replace this with your actual processing logic
    processed_data = data.upper()
    return processed_data

@app.task
def process_data(data):
    # Perform some time-consuming task
    result = perform_processing(data)
    return result
```

*** Flask: Decorating routes and middleware**

In Flask, decorators can be used to decorate routes and middleware functions to add additional functionality or behavior to them. Let's explore how decorators can be used in Flask for routes and middleware.

1. **Decorating Routes**: Decorators can be used to add additional functionality to routes in Flask. Here's an example:

```
from flask import Flask

app = Flask(__name__)

def log_request(func):
    def wrapper(*args, **kwargs):
        # Log the request
        print("Request logged")
        return func(*args, **kwargs)
    return wrapper

@app.route("/")
@log_request
def index():
    return "Hello, World!"

if __name__ == "__main__":
    app.run()
```

In the above example, the `log_request` decorator is defined to log the request before executing the route function. It is then applied to the `index` route using the `@log_request` syntax. When a request is made to the "/" route, the decorator's functionality will be executed before the `index` function is called.

2. **Decorating Middleware:** Middleware functions can be used to intercept and process requests and responses in Flask. Decorators can be used to decorate these middleware functions to add additional behavior.

Here's an example:

```
from flask import Flask, request

app = Flask(__name__)

def log_request_middleware(func):
    def wrapper(*args, **kwargs):
        # Log the request
        print("Request logged")
        return func(*args, **kwargs)
    return wrapper

@app.before_request
@log_request_middleware
def process_request():
    # Perform additional processing on the request
    pass

@app.after_request
def process_response(response):
    # Perform additional processing on the response
    return response

@app.route("/")
def index():
    return "Hello, World!"

if __name__ == "__main__":
    app.run()
```

In the above example, the `log_request_middleware` decorator is defined to log the request before and after it is processed. It is then applied to the `process_request` middleware function using the `@log_request_middleware` syntax. This decorator will be executed before and after each request. Similarly, the `process_response` function is decorated with the `@app.after_request` decorator to process the response.

Decorators provide a powerful way to add functionality to routes and middleware in Flask. They can be used to implement authentication, logging, request/response processing, and many other cross-cutting concerns.

Example :

```
from flask import Flask
```

```python
app = Flask(__name__)

# Route for fingerprint authentication
@app.route("/fingerprint", methods=["POST"])
def fingerprint_authentication():
    # Process fingerprint authentication
    # Use fake data for demonstration
    return "Fingerprint authenticated successfully"

# Route for uploading documents
@app.route("/upload", methods=["POST"])
def upload_document():
    # Process document upload
    # Use fake data for demonstration
    return "Document uploaded successfully"

# Route to submit a form
@app.route("/form", methods=["POST"])
def submit_form():
    # Process form submission
    # Use fake data for demonstration
    return "Form submitted successfully"

# Route to edit a form
@app.route("/form/edit/<form_id>", methods=["POST"])
def edit_form(form_id):
    # Process form editing
    # Use fake data for demonstration
    return f"Form {form_id} edited successfully"

# Route to delete a form
@app.route("/form/delete/<form_id>", methods=["POST"])
def delete_form(form_id):
    # Process form deletion
    # Use fake data for demonstration
    return f"Form {form_id} deleted successfully"

# Route to check medication
@app.route("/medication/<patient_id>")
def check_medication(patient_id):
    # Check medication for the patient
    # Use fake data for demonstration
    return f"Medication information for patient {patient_id}"

# Route to book an appointment
@app.route("/appointment", methods=["POST"])
def book_appointment():
    # Process appointment booking
    # Use fake data for demonstration
    return "Appointment booked successfully"

# Route to check patient information
@app.route("/patient/<patient_id>")
def check_patient_information(patient_id):
    # Check patient information
    # Use fake data for demonstration
    return f"Patient information for patient {patient_id}"

# Route for nurse sign up
@app.route("/signup", methods=["POST"])
def nurse_signup():
    # Process nurse sign up
    # Use fake data for demonstration
    return "Nurse signed up successfully"

# Route to check patient information (accessible only to nurses)
@app.route("/patient/info/<patient_id>")
def check_patient_info_nurse(patient_id):
    # Check patient information (restricted to nurses)
    # Use fake data for demonstration
    return f"Patient information for patient {patient_id}"

if __name__ == "__main__":
    app.run()
```

In the above example, each route is defined using the `@app.route` decorator. The routes are mapped to the specified URLs and HTTP methods. Inside each route function, you can implement the necessary logic to process the request and return a response. The example uses fake data for demonstration purposes, but you can replace it with real data and the actual implementation for each functionality.

Make sure to install Flask and any required dependencies before running the application. You can run the application using `python app.py` or the appropriate command based on your environment.

Note: These examples provide a basic understanding of how decorators can be used in Flask for routes and middleware. In real-world applications, it is recommended to use well-tested libraries and frameworks for more complex functionality, such as authentication and logging.

*** Django: Enhancing view functions with decorators**

In Django, you can enhance view functions using decorators to add additional functionality or modify the behavior of the views. Decorators provide a way to wrap or modify the view functions without modifying their code directly.

Here's an example of how you can enhance view functions with decorators in Django:

```
from django.http import HttpResponse
from django.shortcuts import render
from functools import wraps

# Custom decorator to add additional functionality
def custom_decorator(view_func):
    @wraps(view_func)
    def wrapper(request, *args, **kwargs):
        # Perform some preprocessing or additional logic before executing the view function
        # For example, you can perform authentication or authorization checks here
        if not request.user.is_authenticated:
            return HttpResponse("Unauthorized", status=401)

        # Call the original view function
        response = view_func(request, *args, **kwargs)

        # Perform post-processing or additional logic after executing the view function
        # For example, you can modify the response or perform additional operations

        return response

    return wrapper

# View function enhanced with the custom decorator
@custom_decorator
def my_view(request):
    # View logic goes here
    return render(request, 'my_template.html')
```

In the above example, the `custom_decorator` is a custom decorator function that wraps the original view function `my_view`. It performs some preprocessing before calling the view function, such as checking if the user is authenticated. It also performs post-processing after executing the view function, if needed.

To use the decorator, simply apply it to the view function using the `@decorator_name` syntax, as shown in the example with `@custom_decorator`.

Example

```
from django.contrib.auth.decorators import login_required, permission_required
from django.views.decorators.http import require_POST
from django.core.exceptions import PermissionDenied
from django.http import HttpResponse
from functools import wraps

# Decorator to check if the user is authenticated
def authenticated_required(view_func):
    @wraps(view_func)
    def wrapper(request, *args, **kwargs):
        if not request.user.is_authenticated:
            return HttpResponse("Unauthorized", status=401)
        return view_func(request, *args, **kwargs)
    return wrapper

# Decorator to check if the user has the necessary permissions
def has_permission(permission):
    def decorator(view_func):
        @wraps(view_func)
        def wrapper(request, *args, **kwargs):
            if not request.user.has_perm(permission):
                raise PermissionDenied
            return view_func(request, *args, **kwargs)
        return wrapper
    return decorator

# Decorator to log the view function execution
def log_view(view_func):
    @wraps(view_func)
    def wrapper(request, *args, **kwargs):
        print(f"Executing view: {view_func.__name__}")
        return view_func(request, *args, **kwargs)
    return wrapper

# Decorator to measure the execution time
def measure_execution_time(view_func):
    @wraps(view_func)
    def wrapper(request, *args, **kwargs):
        import time
        start_time = time.time()
        response = view_func(request, *args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Execution time: {execution_time} seconds")
        return response
    return wrapper

# Example usage of decorators
```

```
@login_required
def fingerprint(request):
    # Process fingerprint data
    return HttpResponse("Fingerprint processed successfully")

@permission_required('emr.upload_documents')
def upload_document(request):
    # Upload document logic
    return HttpResponse("Document uploaded successfully")

@require_POST
def submit_form(request):
    # Submit form logic
    return HttpResponse("Form submitted successfully")

@has_permission('emr.edit_form')
def edit_form(request, form_id):
    # Edit form logic
    return HttpResponse(f"Editing form with ID: {form_id}")

@authenticated_required
def delete_information(request, info_id):
    # Delete information logic
    return HttpResponse(f"Deleting information with ID: {info_id}")

@log_view
def check_medication(request):
    # Check medication logic
    return HttpResponse("Medication checked successfully")

@measure_execution_time
def book_appointment(request):
    # Book appointment logic
    return HttpResponse("Appointment booked successfully")

# Apply multiple decorators to a view
@login_required
@has_permission('emr.view_patient_info')
def view_patient_info(request, patient_id):
    # View patient information logic
    return HttpResponse(f"Viewing information for patient with ID: {patient_id}")
```

In the above code, decorators such as `login_required`, `permission_required`, `require_POST`, `authenticated_required`, `has_permission`, `log_view`, and `measure_execution_time` are used to enhance the view functions with additional functionality like authentication, permission checks, logging, execution time measurement, etc.

By using decorators, you can modularize and reuse common functionality across multiple views, keeping your code clean and organized.

*** SQLAlchemy: Extending queries with decorators**

Here's an example of how you can use decorators to extend queries in SQLAlchemy in the context of an EMR (Electronic Medical Records) application for children and nurses:

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()

class Child(Base):
    __tablename__ = 'children'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    age = Column(Integer)
    nurse_id = Column(Integer, ForeignKey('nurses.id'))

    nurse = relationship('Nurse', back_populates='children')

    def __repr__(self):
        return f"<Child(id={self.id}, name={self.name}, age={self.age})>"


class Nurse(Base):
    __tablename__ = 'nurses'

    id = Column(Integer, primary_key=True)
    name = Column(String)

    children = relationship('Child', back_populates='nurse')

    def __repr__(self):
        return f"<Nurse(id={self.id}, name={self.name})>"


# Decorator to filter children by age range
def filter_by_age(min_age, max_age):
    def decorator(func):
        def wrapper(session, *args, **kwargs):
```

```
            query = func(session, *args, **kwargs)
            return query.filter(Child.age >= min_age, Child.age <= max_age)
        return wrapper
    return decorator


# Decorator to filter children by nurse
def filter_by_nurse(nurse_id):
    def decorator(func):
        def wrapper(session, *args, **kwargs):
            query = func(session, *args, **kwargs)
            return query.filter(Child.nurse_id == nurse_id)
        return wrapper
    return decorator


# Usage example
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///emr.db')
Session = sessionmaker(bind=engine)
session = Session()

# Query all children within a certain age range for a specific nurse
@filter_by_age(3, 5)
@filter_by_nurse(1)
def get_children(session):
    return session.query(Child)

children = get_children(session).all()
print(children)
```

In this example, we have two entities: `Child` and `Nurse`, with a one-to-many relationship between them. We define two decorators `filter_by_age` and `filter_by_nurse` that modify a query object by applying additional filtering criteria based on the age range and nurse ID.

The `get_children` function is decorated with both decorators, which means it will retrieve children within a specific age range for a particular nurse. The decorators modify the query object returned by the function to apply the desired filters.

Decorators in Large-Scale Applications and Frameworks

Decorators play a significant role in large-scale applications and frameworks. They provide a powerful mechanism for extending and customizing the behavior of functions, methods, classes, and even entire components in a modular and reusable manner.

8.1 Decorators in Microservices and Serverless Architectures

Decorators play a crucial role in microservices and serverless architectures by enabling code reuse, enhancing functionality, and simplifying common tasks

Example

Here's an example implementation of decorators in a microservice-based EMR application for nurses and children, focusing on authentication and authorization, using fake data:

```
from functools import wraps
from flask import Flask, request, jsonify

app = Flask(__name__)

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Perform authentication logic here
        # Example: Check if the request contains a valid access token
        access_token = request.headers.get('Authorization')
        if not access_token or not validate_access_token(access_token):
            return jsonify({'error': 'Unauthorized'}), 401

        # If authentication is successful, proceed to the decorated function
        return func(*args, **kwargs)

    return wrapper

# Authorization decorator
def authorize(role):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Perform authorization logic here
            # Example: Check if the authenticated user has the required role
            user_role = get_user_role()
            if user_role != role:
                return jsonify({'error': 'Forbidden'}), 403

            # If authorization is successful, proceed to the decorated function
```

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

        return wrapper
    return decorator

# Example usage
@app.route('/patients', methods=['GET'])
@authenticate
def get_patient_records():
    # Retrieve and return patient records
    records = fetch_patient_records()
    return jsonify(records)

@app.route('/patients/<patient_id>', methods=['PUT'])
@authenticate
@authorize(role='nurse')
def update_patient_record(patient_id):
    # Update the specified patient record
    # ...
    return jsonify({'message': 'Patient record updated successfully'})

# Helper functions for authentication and authorization
def validate_access_token(access_token):
    # Perform access token validation
    # In this example, any non-empty access token is considered valid
    return bool(access_token)

def get_user_role():
    # Retrieve the role of the authenticated user
    # In this example, all authenticated users are assigned the role of 'nurse'
    return 'nurse'

def fetch_patient_records():
    # Retrieve patient records from the database
    # In this example, we return fake data for demonstration purposes
    records = [
        {'patient_id': 1, 'name': 'John Doe', 'age': 8, 'gender': 'Male'},
        {'patient_id': 2, 'name': 'Jane Smith', 'age': 5, 'gender': 'Female'},
        # Add more fake patient records here
    ]
    return records

if __name__ == '__main__':
    app.run()
```

In the code above, we have implemented the decorators `authenticate` and `authorize` for authentication and authorization, respectively. The `authenticate` decorator checks the presence and validity of an access token in the request headers, while the `authorize` decorator verifies if the authenticated user has the required role.

We apply these decorators to the routes (`get_patient_records` and `update_patient_record`) to ensure that only authenticated users can access and modify patient records. The implementation includes fake data for demonstration purposes, such as the `fetch_patient_records` function, which returns a list of fake patient records.

Please note that this is a simplified example, and in a real-world application, you would need to implement proper authentication and authorization mechanisms, integrate with a database or data source, and handle more complex business logic based on your specific requirements.

8.2 Applying Decorators in Web Frameworks

Decorators play a significant role in web frameworks, as they provide a convenient way to extend and modify the behavior of web endpoints and views

Example:

```
from functools import wraps
from flask import Flask, request, jsonify

app = Flask(__name__)

# Authentication decorator
def authenticate(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # Check if the request contains an authentication token
        token = request.headers.get('Authorization')
        if not token:
            return jsonify({'error': 'Unauthorized'}), 401

        # Perform authentication logic based on the token
        user_id = verify_token(token)
        if not user_id:
            return jsonify({'error': 'Invalid token'}), 401

        # Add the authenticated user ID to the function arguments
        kwargs['user_id'] = user_id

        # Call the decorated function
        return func(*args, **kwargs)
    return wrapper
```

```python
# Authorization decorator
def authorize(roles):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            # Check if the user has the required role
            user_id = kwargs.get('user_id')
            if not user_id:
                return jsonify({'error': 'Unauthorized'}), 401

            user_role = get_user_role(user_id)
            if user_role not in roles:
                return jsonify({'error': 'Forbidden'}), 403

            # Call the decorated function
            return func(*args, **kwargs)
        return wrapper
    return decorator

# Example routes in the EMR application

@app.route('/patient/<patient_id>', methods=['GET'])
@authenticate
@authorize(['nurse'])
def get_patient_profile(patient_id, user_id):
    # Retrieve patient profile based on patient_id
    # Perform authorization checks to ensure the nurse can access the profile
    # ...
    patient_profile = retrieve_patient_profile(patient_id)
    if not patient_profile:
        return jsonify({'error': 'Patient profile not found'}), 404

    # Return the patient profile
    return jsonify(patient_profile)

@app.route('/appointment/<appointment_id>', methods=['GET'])
@authenticate
def get_appointment_details(appointment_id, user_id):
    # Retrieve appointment details based on appointment_id
    # Perform authorization checks to ensure the user can access the appointment
    # ...
    appointment_details = retrieve_appointment_details(appointment_id)
    if not appointment_details:
        return jsonify({'error': 'Appointment not found'}), 404

    # Return the appointment details
    return jsonify(appointment_details)

# Helper functions for authentication, authorization, and data retrieval

def verify_token(token):
    # Perform token verification logic
    # Return the user ID if the token is valid, otherwise return None
    # ...
    return '12345'  # Example user ID

def get_user_role(user_id):
    # Retrieve the role of the user based on user_id
    # ...
    return 'nurse'  # Example user role

def retrieve_patient_profile(patient_id):
    # Retrieve the patient profile from the database or data source
    # ...
    return {'patient_id': patient_id, 'name': 'John Doe', 'age': 30, ...}

def retrieve_appointment_details(appointment_id):
    # Retrieve the appointment details from the database or data source
    # ...
    return {'appointment_id': appointment_id, 'datetime': '2023-05-30 14:30', ...}

if __name__ == '__main__':
    app.run()
```

In this example, we have implemented decorators for authentication and authorization. The `authenticate` decorator checks for the presence of an authentication token in the request headers, verifies its validity, and adds the authenticated user ID to the function arguments. The `authorize` decorator ensures that only users with specific roles are allowed to access certain routes.

The example includes two routes: `get_patient_profile` and `get_appointment_details`. The `get_patient_profile` route is restricted to authenticated nurses (`@authorize(['nurse'])`), and the `get_appointment_details` route is open to all authenticated users.

By applying these decorators to the routes, we ensure that only authorized users can access sensitive patient information and appointment details. The decorators add an extra layer of security and control to the EMR application.

Note that this is a simplified example for demonstration purposes, and in a real-world EMR application, you would have more complex authentication and authorization mechanisms in place, as well as additional routes and functionality.

8.3 Extending ORM Functionality with Decorators

Extending ORM (Object-Relational Mapping) functionality with decorators can be a powerful approach to enhance database operations and simplify code.

Here's an example of how decorators can be used to extend the ORM functionality in a Flask application:

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///app.db'
db = SQLAlchemy(app)

# Decorator for logging queries
def log_query(func):
    def wrapper(*args, **kwargs):
        query = func(*args, **kwargs)
        app.logger.info(f'Executing query: {query}')
        return query
    return wrapper

# Example model
class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(100))
    email = db.Column(db.String(100))

    @classmethod
    @log_query
    def get_user_by_email(cls, email):
        return cls.query.filter_by(email=email).first()

# Example usage
@app.route('/user/<email>', methods=['GET'])
def get_user(email):
    user = User.get_user_by_email(email)
    if not user:
        return 'User not found', 404
    return f'User: {user.name}, Email: {user.email}'

if __name__ == '__main__':
    app.run()
```

In this example, we define a decorator `log_query` that logs the executed query to the application's logger. We then apply this decorator to the `get_user_by_email` method of the `User` model.

The `log_query` decorator intercepts the execution of the query and logs it using the Flask application's logger. This allows us to easily track and debug the executed queries in the application.

By applying the `log_query` decorator to the `get_user_by_email` method, every time the method is called, the executed query will be logged automatically.

Decorators like this can be used to add additional functionality to ORM methods, such as logging, caching, performance monitoring, or auditing, without modifying the original code or cluttering the model methods.

8.4 Decorators in Data Processing and ETL Pipelines

Decorators can also be used in data processing and ETL (Extract, Transform, Load) pipelines to enhance their functionality and simplify code organization.

Here's an example of how decorators can be applied in data processing and ETL pipelines:

```
def perform_preprocessing(data):
    # Perform preprocessing operations on data
    preprocessed_data = f"Preprocessed {data}"
    return preprocessed_data

def perform_transformation(data):
    # Perform transformation operations on data
    transformed_data = f"Transformed {data}"
    return transformed_data

def perform_loading(data):
    # Perform loading operations on data
    load_status = f"Loaded {data}"
    return load_status

def perform_data_extraction():
    # Simulate data extraction from a data source
    data = "Raw Data"
    return data

def perform_data_processing(data):
    # Perform data processing operations on data
    result = f"Processed {data}"
```

```
    return result

def preprocess_data(func):
    def wrapper(data):
        # Perform preprocessing operations on data
        preprocessed_data = perform_preprocessing(data)
        return func(preprocessed_data)
    return wrapper

def transform_data(func):
    def wrapper(data):
        # Perform transformation operations on data
        transformed_data = perform_transformation(data)
        return func(transformed_data)
    return wrapper

def load_data(func):
    def wrapper(data):
        # Perform loading operations on data
        load_status = perform_loading(data)
        return func(load_status)
    return wrapper

def extract_data():
    # Extract data from a data source
    data = perform_data_extraction()
    return data

@preprocess_data
@transform_data
@load_data
def process_data(data):
    # Process the preprocessed, transformed, and loaded data
    result = perform_data_processing(data)
    return result

# Example usage
data = extract_data()  # Extract data from a data source
result = process_data(data)  # Process the extracted data

print(result)  # Output the result
```

In the above example, we have defined three decorators: `preprocess_data`, `transform_data`, and `load_data`. Each decorator enhances the functionality of the data processing function `process_data` by performing specific operations on the data before and after the function is called.

The `preprocess_data` decorator performs preprocessing operations on the input data, the `transform_data` decorator performs transformation operations, and the `load_data` decorator performs loading operations. The decorators are applied to the `process_data` function using the `@decorator_name` syntax.

When the `process_data` function is called with input data, it goes through each decorator in reverse order of declaration. The data is preprocessed, transformed, and then loaded before being passed to the next decorator. Finally, the processed data is returned as the result.

This allows for a modular and reusable approach to data processing and ETL pipelines. Additional decorators can be added to perform other operations or apply additional functionality as needed.

8.5 Best Practices for Building and Using Decorator-Based Frameworks

When building and using decorator-based frameworks, it's important to follow certain best practices to ensure clean and maintainable code. Here are some best practices to consider:

1. **Use Clear and Descriptive Names**: Choose meaningful names for your decorators and the functions they decorate. This makes the code more readable and understandable.

2. **Document Decorators:** Provide clear documentation for your decorators, explaining their purpose, usage, and any requirements or limitations they have. This helps other developers understand how to use them correctly.

3. **Handle Decorator Arguments**: If your decorators accept arguments, handle them properly. Ensure that the decorator functions and the functions they decorate have compatible signatures and handle the arguments consistently.

4. **Preserve Function Metadata**: When defining decorators, use the `functools.wraps` decorator to preserve the original function's metadata, such as its name, docstring, and module. This helps maintain the integrity and clarity of the decorated function.

5. **Test Decorators:** Write unit tests for your decorators to ensure they work as expected in different scenarios. This helps catch any potential issues or bugs and allows for easier debugging.

6. **Separate Concerns**: Keep your decorators focused on a specific concern or functionality. Avoid creating decorators that perform multiple unrelated tasks. This helps maintain code modularity and makes it easier to understand and modify individual decorators.

7. **Avoid Overusing Decorators:** Use decorators judiciously and avoid overusing them. Too many decorators can make the code complex and hard to follow. Evaluate whether a decorator is truly necessary for a specific functionality before adding it.

8. **Consider Decorator Order**: Pay attention to the order in which decorators are applied, as it can affect the behavior of the decorated functions. Ensure that decorators are applied in the correct order to achieve the desired functionality.

9. **Keep Decorator Logic Simple:** Aim to keep the logic inside decorators simple and focused. Complex logic inside decorators can make the code harder to understand and maintain. Consider extracting complex functionality into separate functions or classes.

10. **Follow Conventions and Style Guidelines**: Adhere to established coding conventions and style guidelines when writing decorator-based frameworks. This helps maintain consistency across the codebase and makes it easier for other developers to contribute or use your framework.

By following these best practices, you can build decorator-based frameworks that are robust, maintainable, and easy to use.

9. Future Trends and Advanced Concepts
9.1 Decorators in Python 3.10 and Beyond

In Python 3.10 and beyond, decorators have been enhanced with the introduction of two new features: parameter specification for decorators and decorator composition.

1. **Parameter Specification for Decorators**: In previous versions of Python, decorators were limited to wrapping a single function without accepting any additional parameters. However, in Python 3.10, you can specify parameters for decorators using the new syntax. This allows decorators to accept arguments, enabling more flexibility and customization.

Example:

```
def my_decorator(param):
    def decorator(func):
        def wrapper(*args, **kwargs):
            # Do something with the parameter
            print(f"Decorator parameter: {param}")
            # Call the decorated function
            return func(*args, **kwargs)
        return wrapper
    return decorator

@my_decorator("Hello")
def my_function():
    print("Inside decorated function")

my_function()  # Output: Decorator parameter: Hello
               #         Inside decorated function
```

2. **Decorator Composition:** Python 3.10 introduces the ability to compose multiple decorators in a concise and readable manner. The new syntax allows decorators to be applied in a stacked fashion, where each decorator modifies the behavior of the function.

Example:

```
def decorator1(func):
    def wrapper(*args, **kwargs):
        print("Decorator 1")
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    def wrapper(*args, **kwargs):
        print("Decorator 2")
        return func(*args, **kwargs)
    return wrapper

@decorator1
@decorator2
def my_function():
    print("Inside decorated function")

my_function()  # Output: Decorator 1
               #         Decorator 2
               #         Inside decorated function
```

These enhancements in Python 3.10 provide more flexibility and expressiveness when working with decorators, allowing for easier customization and composition of decorator functionality. They make it even more powerful to leverage decorators in building reusable and modular code.

9.2 Static Typing and Decorators

Static typing refers to the practice of explicitly specifying the types of variables, function parameters, and function return values at compile time. It helps catch type-related errors early and provides better code understanding and tooling support.

Decorators, on the other hand, are a language feature in Python that allows modifying the behavior of functions or classes by wrapping them with additional code. They are a powerful tool for code reuse, encapsulation, and adding cross-cutting concerns to functions or classes.

Static typing and decorators can work together to provide a more robust and maintainable codebase. By applying static typing to decorator functions and their arguments, you can enforce type checking and ensure type safety for the decorated functions.

To use static typing with decorators, you can follow these steps:

1. Enable static typing in your Python project by using a static type checker like `mypy` or by configuring your IDE to perform static type checking.
2. Define type annotations for the decorator function parameters, specifying the expected types of the decorated function's arguments.
3. Apply the decorator to the target function, ensuring that the decorator's parameters match the annotated types.
4. Run the static type checker to verify the correctness of the type annotations and catch any type-related errors.

Here's an example that demonstrates the usage of static typing with decorators:

```
from typing import Callable

def log_args(func: Callable[..., None]) -> Callable[..., None]:
    def wrapper(*args: int, **kwargs: str) -> None:
        print(f"Arguments: {args}, {kwargs}")
        return func(*args, **kwargs)
    return wrapper

@log_args
def add_numbers(a: int, b: int) -> int:
    return a + b

result = add_numbers(5, 10)
print(result)  # Output: Arguments: (5, 10), {}
               #         15
```

In this example, the `log_args` decorator logs the arguments passed to the decorated function. The static type annotations are added to the decorator function and the decorated function, specifying the expected types of the arguments and return value.

By running a static type checker, such as `mypy`, on this code, you can ensure that the types are correctly defined and catch any type-related errors early in the development process.

Using static typing with decorators helps improve code quality, maintainability, and provides better tooling support for code analysis and refactoring. It enhances the readability and understanding of the codebase and helps prevent type-related bugs.

9.3 Decorators for Concurrency and Asynchronous Programming

Decorators can be powerful tools for managing concurrency and asynchronous programming in Python. They can help simplify the code by abstracting away the complexities of managing concurrent tasks, handling asynchronous operations, and coordinating parallel execution.

Here are some common decorators used in Python for concurrency and asynchronous programming:

1. `@threaded`: This decorator enables executing a function in a separate thread. It is useful for offloading CPU-bound or blocking operations to background threads, keeping the main thread responsive. Here's an example:

```
from concurrent.futures import ThreadPoolExecutor

def threaded(func):
    def wrapper(*args, **kwargs):
        with ThreadPoolExecutor() as executor:
            return executor.submit(func, *args, **kwargs).result()
    return wrapper

@threaded
def compute_square(number):
    return number ** 2
```

```
result = compute_square(5)
print(result)  # Output: 25
```

2. `@asyncio.coroutine` and `@asyncio.gather`: These decorators are used in conjunction with the `asyncio` library for writing asynchronous code using coroutines. They allow defining and executing concurrent or parallel tasks. Here's an example:

```
import asyncio

@asyncio.coroutine
def fetch_url(url):
    # Perform network request asynchronously
    response = yield from aiohttp.request('GET', url)
    return response

@asyncio.coroutine
def process_data(data):
    # Perform data processing asynchronously
    processed_data = yield from some_async_function(data)
    return processed_data

@asyncio.coroutine
def main():
    urls = ['http://example.com', 'http://example.org', 'http://example.net']
    tasks = [fetch_url(url) for url in urls]
    responses = yield from asyncio.gather(*tasks)
    processed_data = yield from process_data(responses)
    print(processed_data)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

3. `@concurrent.futures.as_completed`: This decorator is used with the `concurrent.futures` module to handle multiple concurrent tasks and retrieve results as they complete. It allows executing multiple tasks in parallel and fetching the results in the order of completion.

Here's an example:

```
import concurrent.futures

@concurrent.futures.as_completed
def process_data(data):
    # Perform data processing concurrently
    processed_data = some_function(data)
    return processed_data

def main():
    data = [1, 2, 3, 4, 5]
    with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = [executor.submit(process_data, d) for d in data]
        for future in concurrent.futures.as_completed(futures):
            result = future.result()
            print(result)

if __name__ == '__main__':
    main()
```

These are just a few examples of how decorators can be used to simplify concurrency and asynchronous programming in Python. Depending on the specific requirements and libraries used, decorators can be customized and extended to handle various concurrency models, such as threads, coroutines, or asynchronous event-driven programming.

9.4 Exploring Advanced Decorator Libraries and Ecosystems

When it comes to advanced decorator libraries and ecosystems in Python, there are several popular options available. These libraries provide additional functionalities, patterns, and utilities for working with decorators. Here are a few notable ones:

1. **wrapt:** The `wrapt` library provides a comprehensive and flexible framework for creating and managing decorators. It offers features like automatic function wrapping, support for class decorators, decorator composition, and more.

2. **decorator**: The `decorator` library offers a clean and elegant way to define and use decorators. It provides a decorator decorator (yes, you read that right!) that simplifies the process of creating decorators by handling common tasks such as preserving function signatures and docstrings.

3. **aspectlib:** `aspectlib` is a powerful library for aspect-oriented programming (AOP) in Python. It allows you to apply cross-cutting concerns and behavior to functions and methods using decorators. It supports features like before and after advice, exception handling, and more.

4. **functools:** The `functools` module in the Python standard library offers several decorator-related utilities. For example, `functools.wraps` decorator can be used to update the metadata of a decorated function to match the original function, preserving information like function name, docstring, and parameter signatures.

5. **decorator library ecosystem:** The `decorator` library has an extensive ecosystem of related packages that provide various functionalities and patterns. Some notable packages include `cached-property` for creating cached properties, `contextdecorator` for creating context managers using decorators, `decorator-arguments` for specifying arguments to decorators, and many more.

These libraries and ecosystems provide advanced features and utilities for working with decorators, making it easier to create reusable and powerful decorators. They offer a wide range of functionalities, such as function wrapping, AOP, decorator composition, metadata preservation, and more, allowing you to leverage decorators effectively in your projects.

It's worth exploring these libraries and choosing the ones that best fit your requirements and coding style. Each library may have its own strengths and focus areas, so consider your specific needs and the functionalities provided by these libraries to make an informed decision.

10. Conclusion

10.1 Recap of Advanced Decorator Techniques

Here's a recap of some advanced decorator techniques we discussed:

1. **Function Composition**: Decorators can be used to compose multiple functions together, creating a pipeline of operations where the output of one function becomes the input of the next. This allows for modular and reusable code.

2. **Decorators with Arguments:** Decorators can take arguments to customize their behavior. This allows for greater flexibility and configurability when applying decorators to functions.

3. **Decorator Stacking**: Multiple decorators can be applied to a single function, creating a stacked effect. Each decorator adds its own behavior or functionality to the function, allowing for powerful combinations.

4. **Class-based Decorators**: Decorators can be implemented as classes, providing additional capabilities and statefulness. Class-based decorators can have constructor arguments, instance methods, and maintain internal state.

5. **Decorators as Context Managers:** Decorators can be used as context managers by implementing the `__enter__` and `__exit__` methods. This allows for resource management, exception handling, and context-specific behavior.

6. **Decorator Factories:** Decorators can be defined as higher-order functions that return a decorator function. This enables dynamic generation of decorators with customizable behavior.

7. **Method Decorators:** Decorators can be applied to class methods to modify their behavior. This is useful for adding additional functionality to methods, such as caching, logging, or access control.

8. **Decorators for Class-level Operations:** Decorators can be used to modify class behavior, such as class creation or class attribute modification. This allows for metaprogramming and dynamic class modification.

9. **Decorators in Testing**: Decorators can be used in testing frameworks to add testing-related functionality to functions and methods, such as test case registration, setup, teardown, and assertions.

10. **Decorators in Frameworks and Libraries:** Decorators are widely used in frameworks and libraries to extend functionality, apply middleware, define routes, handle authentication and authorization, and more.

These advanced decorator techniques showcase the power and flexibility of decorators in Python. By leveraging these techniques, you can create elegant, reusable, and customizable code patterns that enhance your applications and frameworks.

In conclusion, decorators are a powerful feature in Python that enable developers to modify and extend the behavior of functions and classes. They offer a flexible and reusable way to add additional functionality, apply cross-cutting concerns, and enhance code modularity. Decorators can be applied in various domains, including web development, data processing, ORM, microservices, and serverless architectures. By leveraging decorators effectively, developers can write cleaner, more maintainable code and tackle complex tasks with ease.