Decorator

A Python decorator is a function that allows you to modify the behavior of another function or method. It's a way to wrap a function with additional functionality without modifying its code directly. Decorators are often used for logging, access control, memoization, and more.

How does a decorator work?

A decorator is typically a function that takes another function as an argument and returns a new function that enhances the original function.

Simple Example:

Let's walk through a simple example.

1. **Define a decorator function**: A decorator is a function that takes another function as an argument and returns a modified version of that function.

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()  # Call the original function
        print("Something is happening after the function is called.")
    return wrapper
```

2. **Define a function to be decorated**: This is the function that you want to enhance with the decorator.

```
def say_hello():
    print("Hello!")
```

3. **Apply the decorator**: You can apply the decorator by either using the `@decorator_name` syntax or manually passing the function into the decorator.

```
@my_decorator
def say_hello():
    print("Hello!")
```

The `@my_decorator` syntax is just shorthand for `say_hello = my_decorator(say_hello)`.

4. **Call the decorated function**: When you call `say_hello()`, it will now run the code inside the decorator first, then the original `say_hello` function.

```
say_hello()
```

Output:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

BluePrint

In Flask, a **Blueprint** is a way to organize your application into reusable components. It allows you to structure your application into distinct sections, each with its own routes, views, and static files. This is especially helpful when building larger applications, as it helps keep the codebase modular and organized.

Key Points about Flask Blueprints:

- **Modular Structure**: Blueprints allow you to define parts of your application (e.g., users, authentication, admin panels) as separate modules that can be registered in the main application.
- **Reusability**: A blueprint can be reused across different Flask apps, making it easier to share functionality between apps.
- **Routing**: You can define routes in a blueprint just like in the main Flask app, and the blueprint will automatically handle those routes when it is registered with the main app.

Example :

1. Define a Blueprint

```
from flask import Blueprint, jsonify

# Create a Blueprint for users
users_bp = Blueprint('users', __name__)
```

Here, a **Blueprint** named `users` is created using `Blueprint('users', __name__)`. The `'users'` is the name of the blueprint, and `__name__` is used to determine the module in which the blueprint is defined. Blueprints are essentially blueprints for routing.

2. Define Routes for the Blueprint

```
@users_bp.route('/users', methods=['GET'])
def list_users():
    users = [
        {"id": 1, "name": "Alice"},
        {"id": 2, "name": "Bob"},
        {"id": 3, "name": "Charlie"}
    ]
    return jsonify({
        "status-text": 'success',
        "status": 200,
        "data": users
    })
```

Here, a route `/users` is defined under the `users_bp` blueprint. When the `/users` endpoint is accessed with a `GET` request, it returns a list of users in JSON format. This is a regular Flask route, but it belongs to the `users` blueprint.

3. Create the Flask Application

```
from flask import Flask, jsonify
from users import users_bp  # Import the users blueprint

# Create the Flask app
app = Flask(__name__)

# Register the users blueprint
app.register_blueprint(users_bp)
```

In this part:

- The Flask application is created with `Flask(__name__)`.
- The `users_bp` blueprint is imported and registered to the app using `app.register_blueprint(users_bp)`.

4. Define Another Route in the Main App

```
@app.route('/hello', methods=['GET'])
def hello():
    return jsonify({
        "status-text": 'success',
        "status": 200,
        "message": "Hello, Flask!"
    })
```

Here, a route `/hello` is defined directly in the main app, which returns a simple JSON response with a greeting message. This route is separate from the blueprint and part of the main Flask app.

5. Run the Application

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

Finally, the Flask app is run with `app.run(debug=True)`. This starts the web server, allowing you to access both `/hello` and `/users` routes.

Structure of the Application:

- The `/hello` route is defined directly in the main Flask application.
- The `/users` route is part of the `users_bp` blueprint, and when this blueprint is registered, its routes are also available.

Advantages of Using Blueprints:

1. **Separation of Concerns**: By using blueprints, you can separate different parts of your application (e.g., users, admin, authentication) into different modules.
2. **Better Organization**: Helps in organizing the code, especially in larger applications, as routes, views, templates, and static files can be grouped by functionality.
3. **Code Reusability**: Blueprints can be reused across different Flask applications, making it easier to share functionality.
