Using Advanced Decorator Workflows

Mateo Prigl
Software Developer

Decorators without Arguments

```
def decorator_function(func):
    def wrapper():
        print("Decorator functionality")
        return func()
    return wrapper


@decorator_function
def some_func():
    pass


# Decorator syntax equals to:
some_func = decorator_function(some_func)
```

Decorators with Arguments

```
def decorator_factory(some_arg):
    def decorator_function(func):
        def wrapper():
            print("Decorator functionality")
            print(f"Some argument value: {some_arg}")
            return func()
        return wrapper
    return decorator_function


@decorator_factory(some_arg="value")
def some_func():
    pass


# Decorator syntax equals to:
some_func = decorator_factory(some_arg="value")(some_func)
```

Decorators with Arguments

```
def decorator_factory(some_arg):
    def decorator_function(func):
        def wrapper():
            print("Decorator functionality")
            print(f"Some argument value: {some_arg}")
            return func()
        return wrapper
    return decorator_function


@decorator_factory(some_arg="value")
def some_func():
    pass


# Decorator syntax equals to:
some_func = decorator_function(some_func)
```

Decorators with Arguments

```
def decorator_function(some_arg):
    def _decorator_function(func):
        def wrapper():
            print("Decorator functionality")
            print(f"Some argument value: {some_arg}")
            return func()
        return wrapper
    return _decorator_function


@decorator_function(some_arg="value")
def some_func():
    pass
```

Applying Multiple Decorators

```
def decorator1(func):
    def wrapper():
        print("Decorator 1 functionality")
        return func()
    return wrapper

def decorator2(func):
    def wrapper():
        print("Decorator 2 functionality")
        return func()
    return wrapper

@decorator1
@decorator2
def some_func():
    pass

# Decorator syntax equals to:
some_func = decorator1(decorator2(some_func))
```

The login_required Decorator

```
def login_required(f):
    @wraps(f)
    def _login_required(*args, **kwargs):
        if current_user.is_anonymous():
            flash("You need to be logged in to access this page", "danger")
            return redirect(url_for("auth.login"))
        return f(*args, **kwargs)
    return _login_required


@auth.route("/logout")
@login_required
def logout():
    # ...
```

The role_required Decorator

```
def role_required(role):
    def _role_required(f):
        @wraps(f)
        def __role_required(*args, **kwargs):
            if not current_user.is_role(role):
                flash("You are not authorized to access this page", "danger")
                return redirect(url_for("main.home"))
            return f(*args, **kwargs)
        return __role_required
    return _role_required
```

```
@gig.route("/create", methods=["GET", "POST"])
@login_required
@role_required(Role.EMPLOYER)
def create():
    # ...
```

Summary

Decorators can receive arguments

Multiple decorators can be applied to a single function

Frameworks like Flask make use of decorators to implement utilities

Up Next:

Decorating Classes and Class Decorators