Python-Decorators

Decorators in Python

In Python, a decorator is a design pattern that allows you to modify the behavior of a function or class without changing its implementation. It's a powerful tool that enables you to add functionality to an existing function or class without modifying its source code.

Why use Decorators?

Decorators are useful when you want to:

- Log function calls and their results
- Implement authentication and authorization checks
- Time functions for performance measurement
- Retry failed operations

Basic Syntax of a Decorator

A decorator is defined using the `@` symbol followed by the name of the decorator. The basic syntax is:

```
 @decorator_name
def function_to_decorate():
    pass
```

Here's an example of a simple decorator that logs function calls and their results:

```
 import logging

logging.basicConfig(level=logging.INFO)

def log_call(func):
    def wrapper(*args, **kwargs):
        logging.info(f"Calling {func.__name__} with args: {args}, kwargs: {kwargs}")
        result = func(*args, **kwargs)
        logging.info(f"{func.__name__} returned: {result}")
        return result
    return wrapper

@log_call
def add(a, b):
    return a + b

print(add(2, 3))
```

In this example:

- We define the `log_call` decorator, which takes a function as an argument.
- The `wrapper` function is defined inside `log_call`, which calls the original function and logs its results.
- We apply the `@log_call` decorator to the `add` function, which modifies its behavior.

Example Use Cases

1. **Logging**: Log function calls and their results:

```
 @log_call
def divide(a, b):
    return a / b

print(divide(10, 2))
```

Output:

```
 INFO:root:Calling divide with args: (10, 2), kwargs: {}
INFO:root:divide returned: 5.0
```

2. **Authentication**: Check if the user is authenticated before allowing function calls:

```
 def auth_required(func):
    def wrapper(*args, **kwargs):
        if not is_authenticated():
            raise AuthenticationError("User is not authenticated")
        return func(*args, **kwargs)
    return wrapper

@auth_required
def sensitive_data():
    # Only accessible to authenticated users
    pass

try:
    sensitive_data()
except AuthenticationError as e:
    print(e)
```

3. **Retry**: Retry failed operations with a specified number of attempts:

```
 import random

def retry(max_attempts):
    def decorator(func):
        def wrapper(*args, **kwargs):
            attempts = 0
            while attempts < max_attempts:
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    attempts += 1
                    print(f"Attempt {attempts} failed: {e}")
            raise Exception("Max attempts reached")
        return wrapper
    return decorator

@retry(3)
def example_function():
    # Simulate a random failure
    if random.random() < 0.5:
        raise Exception("Random failure")
    return "Success"

print(example_function())
```

These examples demonstrate the power and flexibility of decorators in Python.

Curated by Brajesh Kumar