Python Decorators Exercises with Solutions

ali.zainoul.az@gmail.com

November 26, 2025

Contents

Preface

This document contains 10 exercises about Python decorators, ordered from simplest to most advanced. Each exercise includes: **Objective**, **Statement**, and **Solution**.

1 Exercise 1 Simple decorator

Objective: understand the minimal structure of a decorator.

Statement Write a decorator `hello` that prints “Hello!” before executing the function.

Solution

```python
def hello(func):
    def wrapper():
        print("Hello!")
        return func()
    return wrapper

@hello
def say_something():
    print("I am a function.")

say_something()
```

2 Exercise 2 Decorator modifying return value

Objective: intercept and modify return value.

Statement Create a decorator `double_return` that multiplies the returned value by 2.

Solution

```python
def double_return(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result * 2
    return wrapper

@double_return
def give_x(x):
    return x

print(give_x(5))  # 10
```

3 Exercise 3 Decorator with arguments

Objective: handle `*args` and `**kwargs`.

Statement Write a decorator `log` that prints function name and arguments.

Solution

```python
def log(func):
    def wrapper(*args, **kwargs):
        print(f"Call of function: {func.__name__}")
        print(f"Arguments: {args} {kwargs}")
        return func(*args, **kwargs)
    return wrapper

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

add(3, 4)
```

4 Exercise 4 Preserve name and docstring

Objective: use `functools.wraps`.

Statement Improve the previous decorator so the function keeps its original name and docstring.

Solution

```python
from functools import wraps

def log(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f"Call: {func.__name__}")
        return func(*args, **kwargs)
    return wrapper

@log
def add(a, b):
    """Add two numbers."""
    return a + b

print(add.__name__)   # add
print(add.__doc__)    # Add two numbers.
```

5 Exercise 5 Parameterized decorator

Objective: make a decorator that takes parameters.

Statement Create a decorator `repeat(n)` that runs the function `n` times.

Solution

```python
def repeat(n):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(n):
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def greet():
    print("Hi!")

greet()
```

6 Exercise 6 Argument type checking

Objective: validate argument types.

Statement Create decorator `type_int` that ensures all positional arguments are integers.

Solution

```
def type_int(func):
    def wrapper(*args, **kwargs):
        for a in args:
            if not isinstance(a, int):
                raise TypeError("All positional arguments must be int")
        return func(*args, **kwargs)
    return wrapper

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

print(add(3, 5))
```

7 Exercise 7 Memoization (caching)

Objective: implement a simple cache.

Statement Create decorator `memo` that saves results in a cache.

Solution

```
def memo(func):
    cache = {}
    def wrapper(*args):
        if args in cache:
            return cache[args]
        res = func(*args)
        cache[args] = res
        return res
    return wrapper

@memo
def square(n):
    print("Computing...")
    return n * n

print(square(4))
print(square(4))  # uses cache
```

8 Exercise 8 Timing decorator

Objective: measure execution time.

Statement Create decorator `timing` that prints execution duration.

Solution

```
import time

```

```python
def timing(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"Time: {end - start:.5f}s")
        return result
    return wrapper

@timing
def long_task():
    time.sleep(1)

long_task()
```

9 Exercise 9 Decorator for instance methods

Objective: manage `self` correctly.

Statement Create a decorator `debug` that prints method name and class name.

Solution

```python
def debug(method):
    def wrapper(self, *args, **kwargs):
        print(f"Calling {method.__name__} in {self.__class__.__name__}")
        return method(self, *args, **kwargs)
    return wrapper

class Test:
    @debug
    def method(self):
        print("Running...")

t = Test()
t.method()
```

10 Exercise 10 Parameterized method decorator

Objective: combine parameterization and method decoration.

Statement Create a decorator `authorize(roles)` that checks whether `self.user.role` belongs to allowed roles.

Solution

```python
def authorize(roles):
    def decorator(method):
        def wrapper(self, *args, **kwargs):
            if self.user.role not in roles:
                raise PermissionError("Access denied")
            return method(self, *args, **kwargs)
        return wrapper
    return decorator

class User:
    def __init__(self, role):
        self.role = role

class App:
    def __init__(self, user):
        self.user = user

    @authorize(["admin"])
    def delete(self):
        print("Deletion performed.")

app = App(User("admin"))
app.delete()
```