Chapter 7 - Closures and Decorators

1. Introduction to Closures

A **closure** is a function that has access to variables from its enclosing scope, even after the outer function has finished execution. It allows a function to remember and access the environment in which it was created.

1.1 How Closures Work

A closure occurs when a nested function refers to a variable from its enclosing function, and the outer function has finished execution. This is often used to implement functions with state or configuration that should be remembered across multiple calls.

1.2 Example of a Closure

Here's a simple example to demonstrate how closures work:

```
def outer_function(x):
    def inner_function(y):
        return x + y  # 'x' is captured by the closure
    return inner_function

# Create a closure where x is remembered
add_five = outer_function(5)
print(add_five(10))  # Output: 15
```

In this example, `inner_function` is a closure because it references `x`, which is defined in the outer scope of `outer_function`. Even after `outer_function` finishes execution, `inner_function` still "remembers" the value of `x`.

---

2. Practical Use of Closures

Closures can be particularly useful for tasks such as:

- **Function Configuration**: You can create customized functions with different behaviors by passing parameters to the outer function and returning a function with those parameters fixed.

- **Data Encapsulation**: Closures help encapsulate data and prevent it from being accessed or modified directly from outside.

2.1 Example: Using Closures for Configuration

```
def make_multiplier(factor):
    def multiplier(number):
        return number * factor
    return multiplier

# Create a multiplier that always multiplies by 3
multiply_by_3 = make_multiplier(3)
print(multiply_by_3(5))  # Output: 15

# Create a multiplier that always multiplies by 5
multiply_by_5 = make_multiplier(5)
print(multiply_by_5(5))  # Output: 25
```

In this example, `make_multiplier` is a factory function that returns a closure `multiplier` which remembers the `factor` it was created with.

---

3. Introduction to Decorators

A **decorator** is a design pattern in Python that allows you to modify or extend the behavior of functions or methods without changing their actual code. Decorators are often used to wrap a function and add functionality to it, like logging, timing, or authorization checks.

3.1 How Decorators Work

A decorator is essentially a function that takes another function as an argument and returns a new function that typically extends the behavior of the original function.

3.2 Syntax of a Decorator

The basic syntax of a decorator in Python is as follows:

```
def decorator_function(original_function):
    def wrapper_function():
        print("Wrapper executed this before
{}".format(original_function.__name__))
```

```
        return original_function()
    return wrapper_function
```

You can apply the decorator to a function using the `@` symbol:

```
@decorator_function
def display():
    return "Display function executed"

# Calling the decorated function
print(display())
```

In this case, `decorator_function` wraps the `display` function, modifying its behavior.

---

4. Practical Use of Decorators

Decorators are commonly used in:

- **Logging**: Adding logging to functions automatically without modifying their code.
- **Authentication**: Checking if a user is authorized before executing a function (e.g., in web applications).
- **Memoization**: Caching the results of function calls to improve performance.

4.1 Example: Simple Logging Decorator

Here's an example of a simple decorator that logs the function name and its arguments:

```
def log_function_call(func):
    def wrapper(*args, **kwargs):
        print(f"Function {func.__name__} was called with arguments {args} and
keyword arguments {kwargs}")
        return func(*args, **kwargs)
    return wrapper

# Applying the decorator
@log_function_call
def add(a, b):
    return a + b

# Calling the decorated function
print(add(5, 3))
```

Output:

```
Function add was called with arguments (5, 3) and keyword arguments {}
8
```

The decorator `log_function_call` adds logging functionality to the `add` function without modifying its original code.

---

5. Decorators with Arguments

Sometimes, you might want to pass arguments to your decorators. To do this, you need to create a decorator factory function that takes arguments and returns a decorator.

5.1 Example: Parameterized Decorator

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

@repeat(3)
def greet(name):
    print(f"Hello, {name}")

greet("Alice")
```

Output:

```
Hello, Alice
Hello, Alice
Hello, Alice
```

In this example, the `repeat` decorator takes an argument `n` and repeats the execution of the decorated function `n` times.

---

6. Using Decorators in Django

In Django, decorators are widely used for tasks like:

- **View permissions**: Check if the user has permission to access a view.
- **Caching**: Cache the response of a view function.
- **CSRF protection**: Automatically add protection to POST requests.

6.1 Example: Django View Decorators

Here's how decorators are used in Django views:

```
from django.http import HttpResponse
from django.contrib.auth.decorators import login_required

@login_required
def my_view(request):
    return HttpResponse("This is a private view")

# The @login_required decorator ensures that only authenticated users can
access the view.
```

In this example, the `@login_required` decorator checks whether the user is logged in before executing the view function.

---

7. Exercise

Exercise 1: Implementing a Simple Closure

Create a closure called `make_adder` that takes an integer `x` and returns a function that adds `x` to another number `y`.

```
def make_adder(x):
    def adder(y):
        return x + y
    return adder

# Test the closure
add_10 = make_adder(10)
print(add_10(5))  # Output: 15
```

Exercise 2: Creating a Decorator

Write a decorator that logs the time a function takes to execute.

```
import time

def log_execution_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"Function {func.__name__} took {end_time - start_time:.4f}
seconds to execute.")
        return result
    return wrapper

@log_execution_time
def some_long_function():
    time.sleep(2)  # Simulate a delay

some_long_function()
```

Exercise 3: Applying Decorators in Django

Create a custom decorator that checks if a user is an admin before allowing access to a view. If the user is not an admin, redirect them to the login page.

```
from django.shortcuts import redirect
from django.http import HttpResponse

def admin_required(view_func):
    def wrapper(request, *args, **kwargs):
        if not request.user.is_superuser:
            return redirect('login')
        return view_func(request, *args, **kwargs)
    return wrapper

# Applying the decorator to a view
@admin_required
```

```
def admin_dashboard(request):
    return HttpResponse("Welcome to the admin dashboard!")
```