Decorators

Decorators are functions that modify or enhance the behavior of other functions or methods without changing their source code directly. Decorators use the `@decorator_name` syntax before a function definition to apply the decorator to that function. Decorators are commonly used for tasks such as logging, caching, authentication, and more.

Example:

```
def my_decorator(func):
 def wrapper(*args, **kwargs):
 print("Before function call")
 result = func(*args, **kwargs)
 print("After function call")
 return result
 return wrapper

@my_decorator
def my_function():
 print("Inside function")

my_function()
```

In this example, `my_decorator` is a decorator that adds behavior before and after the function call.

The workflow of a decorator involves several steps that occur when a decorated function is defined and called. Here's a breakdown of the decorator workflow in Python:

Decorator Definition:

- A decorator is defined as a regular Python function that takes another function as its argument and returns a new function.
- Inside the decorator function, a wrapper function is typically defined to modify or enhance the behavior of the original function.

```
def my_decorator(func):
 def wrapper(*args, **kwargs):
 # Add functionality before calling the original function
 print("Before function call")
```

```python
result = func(*args, **kwargs)
# Add functionality after calling the original function
print("After function call")
return result
return wrapper
```

Applying the Decorator:

- To apply the decorator to a function, use the `@decorator_name` syntax before the function definition.
- This syntax tells Python to pass the function to the decorator and reassign the function to the result returned by the decorator.

```python
def my_function():
 print("Inside function")
```

Function Call:

- When the decorated function is called, Python executes the wrapper function instead of the original function.
- Inside the wrapper function, the original function is called, and any additional functionality defined in the decorator is executed.

```python
my_function()
```

Execution Flow:

- The execution flow proceeds as follows:
    - When `my_function()` is called, Python executes `wrapper()` instead.
    - Inside `wrapper()`, any code defined before the call to the original function is executed (e.g., printing "Before function call").
    - The original function (`my_function`) is called with the provided arguments.
    - Any additional functionality defined after the call to the original function is executed (e.g., printing "After function call").
    - The result of the original function call is returned to the caller.

In summary, the workflow of a decorator involves defining a decorator function, applying it to a function using the `@` syntax, and executing the wrapper function instead of the original function when called. The wrapper function modifies or enhances the behavior of the original function as defined in the decorator, and the result is returned to the caller.

Exercises and Answers for Decorators

Exercise 1:
Create a decorator called `debug` that prints the name of the function being called along with its arguments and return value. Apply the `debug` decorator to a function called `add` that adds two numbers together. Test the `add` function with different arguments and observe the debug output.

Answer 1:

```python
def debug(func):
 def wrapper(*args, **kwargs):
 print(f"Calling {func.__name__} with args: {args}, kwargs: {kwargs}")
 result = func(*args, **kwargs)
 print(f"{func.__name__} returned: {result}")
 return result
 return wrapper

@debug
def add(x, y):
 return x + y

# Test the function
result = add(3, 5)
print("Result:", result)
```

Exercise 2:
Write a decorator called `timer` that measures the execution time of a function. Apply the `timer` decorator to a function called `fibonacci` that calculates the nth Fibonacci number recursively. Test the `fibonacci` function with different values of `n` and observe the execution time.

Answer 2:

```python
import time

def timer(func):
 def wrapper(*args, **kwargs):
 start_time = time.time()
 result = func(*args, **kwargs)
 end_time = time.time()
 print(f"{func.__name__} executed in {end_time - start_time:.6f} seconds")
 return result
 return wrapper
```

```python
@timer
def fibonacci(n):
 if n <= 1:
 return n
 else:
 return fibonacci(n-1) + fibonacci(n-2)

# Test the function
result = fibonacci(10)
print("Result:", result)
```