DECORATORS IN PYTHON

Presented By : Ihor Afanasiev
ARAI 7|Becode

WHAT ARE DECORATORS?

Definition:

A **decorator** is a function that takes another function as input, enhances or modifies its behavior, and returns the modified function.

Core Concept:

"Wrapping" a function with additional functionality without altering its original code.

Key Features:

- Clean and reusable code.
- Applied using the @decorator_name syntax.

WHY USE DECORATORS?

Advantages of Decorators:

- **Code Reusability:** Write functionality once and apply it across multiple functions.
- **Separation of Concerns:** Keep core logic and auxiliary logic separate.
- **Dynamic Behavior:** Modify behavior of functions at runtime.

Common Use Cases:

1. **Logging**
2. **Access control (e.g., authentication/authorization)**
3. **Input validation**
4. **Performance measurement (e.g., timing)**
5. **Memoization and caching**

CUSTOM DECORATORS

Basic Example

A decorator wraps the original function and modifies its behavior.

```python
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} executed in {end - start:.4f} seconds.")
        return result
    return wrapper

@timer
def calculate_sum(n):
    return sum(range(n))

calculate_sum(10**6)
```

A decorator in Python is a higher-order function. It can be broken down into three key components:

- The decorator function — wraps the original function.
- The wrapper — a function inside the decorator that performs additional logic.
- Using @decorator — syntactic sugar for applying the decorator.

EXAMPLES OF BUILT-IN DECORATORS

@functools.wraps

- Ensures that the decorated function retains its original metadata (e.g., name, docstring, etc.).
- Useful when creating custom decorators.

```python
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Wrapper executed!")
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def greet():
    """This function greets the user."""
    print("Hello!")

print(greet.__name__)  # Outputs: greet
print(greet.__doc__)   # Outputs: This function greets the user.
```

EXAMPLES OF BUILT-IN DECORATORS

@functools.wraps

- Ensures that the decorated function retains its original metadata (e.g., name, docstring, etc.).
- Useful when creating custom decorators.

```python
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("Wrapper executed!")
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def greet():
    """This function greets the user."""
    print("Hello!")

print(greet.__name__)  # Outputs: greet
print(greet.__doc__)   # Outputs: This function greets the user.
```

@property

- Allows methods to be accessed like attributes.

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height

rect = Rectangle(4, 5)
print(rect.area)  # Outputs: 20
```

CUSTOM DECORATORS

Parameterized Example

You can pass arguments to a decorator by adding a wrapper for the decorator itself.

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

@repeat(3)
def say_hello():
    print("Hello!")

say_hello()
```

REAL-WORLD EXAMPLES OF DECORATORS

LOGGING DECORATOR:

```
def log(func):
    def wrapper(*args, **kwargs):
        print(f"Function {func.__name__} called with {args}, {kwargs}")
        return func(*args, **kwargs)
    return wrapper

@log
def process_data(data):
    print(f"Processing {data}")

process_data("data.csv")
```

REAL-WORLD EXAMPLES OF DECORATORS

AUTHENTICATION DECORATOR:

```
def authenticate(user_role):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if user_role != "admin":
                print("Access Denied!")
            else:
                return func(*args, **kwargs)
        return wrapper
    return decorator

@authenticate("user")
def delete_record():
    print("Record deleted.")

delete_record()
```

REAL-WORLD EXAMPLES OF DECORATORS

RETRY FAILED FUNCTION CALLS:

```python
import time
from functools import wraps

def retry(times, delay=1):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for i in range(times):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    print(f"Retry {i+1}/{times} failed: {e}")
                    time.sleep(delay)
            raise Exception("Function failed after retries")
        return wrapper
    return decorator

@retry(times=3, delay=2)
def unstable_function():
    if time.time() % 2 > 1.5:  # Randomly fail
        raise ValueError("Random failure")
    return "Success!"

print(unstable_function())
```

CHAINING MULTIPLE DECORATORS

```python
def uppercase(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs).upper()
    return wrapper

def exclaim(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs) + "!"
    return wrapper

@uppercase
@exclaim
def greet(name):
    return f"Hello, {name}"

print(greet("Alice"))  # HELLO, ALICE!
```

DECORATORS IN CLASSES

```python
def check_positive(func):
    def wrapper(self, value):
        if value < 0:
            raise ValueError("Value must be positive!")
        return func(self, value)
    return wrapper

class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance

    @check_positive
    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited {amount}, new balance: {self.balance}")

account = BankAccount()
account.deposit(100)
# account.deposit(-50)  # Raises ValueError
```

CLASS-BASED DECORATORS

A decorator can also be implemented as a class with the **__call__** method.

```python
class CallCounter:
    def __init__(self, func):
        self.func = func
        self.call_count = 0  # State to track the number of calls

    def __call__(self, *args, **kwargs):
        self.call_count += 1  # Increment the call count
        print(f"Call {self.call_count} to {self.func.__name__}")
        return self.func(*args, **kwargs)

@CallCounter
def greet(name):
    print(f"Hello, {name}!")

# Using the decorated function
greet("Alice")
greet("Bob")
greet("Charlie")
```

- **Initialization (__init__):** The decorator class takes the target function as an argument and stores it in the instance (e.g., self.func).

- **Callability (__call__):** The __call__ method makes the class instance callable, similar to a function. It allows adding logic before and after invoking the wrapped function.

RESOURCES

- DataCamp tutorial
- Python documentation
- Python build-in decorators library

TRY IT YOURSELF!