**Python Decorators**

**Introduction to Decorators**

**Definition**: Decorators are a powerful and flexible way to modify the behavior of functions or methods. They allow you to wrap another function to extend its behavior without permanently modifying it.

**Syntax**: A decorator is applied to a function by prefixing the function definition with the @decorator\_name syntax.

**Basic Example**

1. **Without Decorators**:

def say\_hello():

return "Hello!"

def greet\_decorator(func):

def wrapper():

return f"Greetings! {func()}"

return wrapper

say\_hello = greet\_decorator(say\_hello)

print(say\_hello()) # Output: Greetings! Hello!

1. **With Decorators**:

def greet\_decorator(func):

def wrapper():

return f"Greetings! {func()}"

return wrapper

@greet\_decorator

def say\_hello():

return "Hello!"

print(say\_hello()) # Output: Greetings! Hello!

**Decorator Functions**

**Creating a Decorator**:

1. **Basic Structure**:

def my\_decorator(func):

def wrapper():

# Code to execute before the function call

result = func()

# Code to execute after the function call

return result

return wrapper

1. **Applying the Decorator**:

@my\_decorator

def my\_function():

print("This is my function.")

my\_function()

**Example: Logging Decorator**:

def logging\_decorator(func):

def wrapper(\*args, \*\*kwargs):

print(f"Calling function: {func.\_\_name\_\_}")

result = func(\*args, \*\*kwargs)

print(f"Function {func.\_\_name\_\_} returned {result}")

return result

return wrapper

@logging\_decorator

def add(a, b):

return a + b

print(add(5, 3)) # Output:

# Calling function: add

# Function add returned 8

# 8

**Decorators with Arguments**

1. **Decorator Function with Arguments**:

def repeat(num\_times):

def decorator(func):

def wrapper(\*args, \*\*kwargs):

result = None

for \_ in range(num\_times):

result = func(\*args, \*\*kwargs)

return result

return wrapper

return decorator

@repeat(num\_times=3)

def say\_hello():

print("Hello!")

say\_hello()

**Built-in Decorators**

1. **@property**:
   * Converts a method into a property.

class Person:

def \_\_init\_\_(self, name):

self.\_name = name

@property

def name(self):

return self.\_name

@name.setter

def name(self, value):

self.\_name = value

person = Person("Alice")

print(person.name) # Output: Alice

person.name = "Bob"

print(person.name) # Output: Bob

1. **@staticmethod and @classmethod**:
   * @staticmethod: Defines a method that does not operate on an instance or class.
   * @classmethod: Defines a method that operates on the class rather than instances.

class MyClass:

@staticmethod

def static\_method():

print("This is a static method.")

@classmethod

def class\_method(cls):

print(f"This is a class method of {cls}.")

MyClass.static\_method() # Output: This is a static method.

MyClass.class\_method() # Output: This is a class method of <class '\_\_main\_\_.MyClass'>.

**Chaining Decorators**

Multiple decorators can be applied to a single function by stacking them:

def decorator\_one(func):

def wrapper(\*args, \*\*kwargs):

print("Decorator One")

return func(\*args, \*\*kwargs)

return wrapper

def decorator\_two(func):

def wrapper(\*args, \*\*kwargs):

print("Decorator Two")

return func(\*args, \*\*kwargs)

return wrapper

@decorator\_one

@decorator\_two

def say\_hello():

print("Hello!")

say\_hello() # Output:

# Decorator One

# Decorator Two

# Hello!

**Practical Examples**

1. **Timing Function Execution**:

import time

def timer\_decorator(func):

def wrapper(\*args, \*\*kwargs):

start\_time = time.time()

result = func(\*args, \*\*kwargs)

end\_time = time.time()

print(f"Execution time: {end\_time - start\_time} seconds")

return result

return wrapper

@timer\_decorator

def long\_running\_function():

time.sleep(2)

long\_running\_function() # Output: Execution time: 2.0... seconds

1. **Access Control**:

def requires\_authentication(func):

def wrapper(user, \*args, \*\*kwargs):

if not user.is\_authenticated:

raise PermissionError("User is not authenticated.")

return func(user, \*args, \*\*kwargs)

return wrapper

class User:

def \_\_init\_\_(self, name, authenticated):

self.name = name

self.is\_authenticated = authenticated

@requires\_authentication

def view\_dashboard(user):

return f"Welcome to the dashboard, {user.name}!"

user = User("Alice", True)

print(view\_dashboard(user)) # Output: Welcome to the dashboard, Alice!

unauthenticated\_user = User("Bob", False)

print(view\_dashboard(unauthenticated\_user)) # Raises PermissionError

**Summary**

* **Decorators** are a powerful way to extend or modify the behavior of functions and methods.
* They can be used for logging, access control, timing functions, and more.
* Decorators can be stacked and can accept arguments for more flexible behavior.

Understanding decorators is crucial for writing clean, reusable, and efficient code in Python.