**Python Decorators Overview**

decorator method example and use cases

A \*\*decorator\*\* in Python is a design pattern that allows you to modify or extend the behavior of a function or method without changing its code. Decorators are often used for logging, enforcing access control, instrumentation, caching, and more.  
  
Here’s an example of a decorator method and its use cases:  
  
---  
  
### \*\*Example of a Decorator Method\*\*  
  
```python  
# A simple decorator to log the execution of a function  
def log\_decorator(func):  
 def wrapper(\*args, \*\*kwargs):  
 print(f"Function '{func.\_\_name\_\_}' is being called with arguments: {args} {kwargs}")  
 result = func(\*args, \*\*kwargs)  
 print(f"Function '{func.\_\_name\_\_}' returned: {result}")  
 return result  
 return wrapper  
  
# Applying the decorator to a function  
@log\_decorator  
def add(a, b):  
 return a + b  
  
# Using the decorated function  
result = add(5, 3)  
print(f"Result: {result}")  
```  
  
\*\*Output:\*\*  
```  
Function 'add' is being called with arguments: (5, 3) {}  
Function 'add' returned: 8  
Result: 8  
```  
  
---  
  
### \*\*Use Cases of Decorators\*\*  
  
1. \*\*Logging\*\*:  
 Decorators are commonly used to log function calls, arguments, and return values.  
  
 ```python  
 def log(func):  
 def wrapper(\*args, \*\*kwargs):  
 print(f"Calling {func.\_\_name\_\_} with {args} and {kwargs}")  
 return func(\*args, \*\*kwargs)  
 return wrapper  
 ```  
  
2. \*\*Access Control\*\*:  
 Decorators can enforce user authentication or permissions before executing a function.  
  
 ```python  
 def requires\_authentication(func):  
 def wrapper(\*args, \*\*kwargs):  
 user\_authenticated = True # Example condition  
 if not user\_authenticated:  
 raise PermissionError("User not authenticated")  
 return func(\*args, \*\*kwargs)  
 return wrapper  
 ```  
  
3. \*\*Caching\*\*:  
 Decorators can cache the results of expensive function calls to improve performance.  
  
 ```python  
 def cache(func):  
 results = {}  
 def wrapper(\*args):  
 if args in results:  
 return results[args]  
 results[args] = func(\*args)  
 return results[args]  
 return wrapper  
 ```  
  
4. \*\*Timing\*\*:  
 Measure the execution time of a function.  
  
 ```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:.4f} seconds")  
 return result  
 return wrapper  
 ```  
  
5. \*\*Validation\*\*:  
 Validate inputs to a function.  
  
 ```python  
 def validate\_inputs(func):  
 def wrapper(\*args, \*\*kwargs):  
 if any(arg < 0 for arg in args):  
 raise ValueError("All arguments must be non-negative")  
 return func(\*args, \*\*kwargs)  
 return wrapper  
  
 @validate\_inputs  
 def multiply(a, b):  
 return a \* b  
  
 print(multiply(3, 4)) # Works  
 print(multiply(-1, 4)) # Raises ValueError  
 ```  
  
6. \*\*Class Methods and Static Methods\*\*:  
 Decorators like `@classmethod` and `@staticmethod` are built-in Python decorators used to define methods that are bound to the class rather than an instance.  
  
 ```python  
 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()  
 MyClass.class\_method()  
 ```  
  
---  
  
### \*\*Chaining Multiple Decorators\*\*  
  
You can apply multiple decorators to a single function. They are applied from the bottom up.  
  
```python  
def decorator1(func):  
 def wrapper(\*args, \*\*kwargs):  
 print("Decorator 1")  
 return func(\*args, \*\*kwargs)  
 return wrapper  
  
def decorator2(func):  
 def wrapper(\*args, \*\*kwargs):  
 print("Decorator 2")  
 return func(\*args, \*\*kwargs)  
 return wrapper  
  
@decorator1  
@decorator2  
def greet(name):  
 print(f"Hello, {name}!")  
  
greet("Alice")  
```  
  
\*\*Output:\*\*  
```  
Decorator 1  
Decorator 2  
Hello, Alice!  
```  
  
---  
  
### \*\*Key Points\*\*  
- Decorators are a powerful way to modify or extend the behavior of functions or methods.  
- They are widely used in frameworks like Flask and Django for routing, middleware, and more.  
- Python provides built-in decorators like `@staticmethod`, `@classmethod`, and `@property`.