Decorators در پایتون

در پایتون، **decorator** یک الگو (pattern) است که برای تغییر یا بهبود رفتار توابع یا کلاس‌ها بدون نیاز به تغییر کد اصلی آن‌ها استفاده می‌شود. به‌عبارت ساده‌تر، یک decorator به شما این امکان را می‌دهد که رفتار یک تابع را در حین اجرا تغییر دهید، بدون اینکه نیاز به تغییر در خود تابع باشد.

---

1. معرفی مفهوم Decorator

یک **decorator** در پایتون به‌طور معمول یک تابع است که به یک تابع دیگر اعمال می‌شود و تابع اصلی را تغییر می‌دهد یا ویژگی‌های اضافی به آن اضافه می‌کند.

ساختار کلی یک Decorator

در اینجا یک مثال ساده از decorator آورده شده است:

```
# تعریف یک decorator ساده
def my_decorator(func):
    def wrapper():
        print("قبل از اجرای تابع")
        func()
        print("بعد از اجرای تابع")
    return wrapper

# استفاده از decorator
@my_decorator
def hello():
    print("سلام!")

# فراخوانی تابع
hello()
```

در این مثال، `my_decorator` تابع `hello` را به‌طور غیرمستقیم تغییر می‌دهد تا قبل و بعد از اجرای `()hello` پیامی چاپ کند.

خروجی:

```
قبل از اجرای تابع
سلام!
بعد از اجرای تابع
```

در اینجا، `my_decorator@` به‌طور خودکار به تابع `hello` افزوده شده است. این کار باعث می‌شود که قبل و بعد از اجرای تابع، پیامی چاپ شود.

---

2. کاربرد Decorator برای مدیریت ویژگی‌ها

الف. مدیریت اعتبارسنجی (Validation)

یک کاربرد رایج decorator، اعتبارسنجی ورودی‌های تابع است. مثلا می‌توانیم بررسی کنیم که آیا ورودی‌های تابع معتبر هستند یا خیر.

```
def validate_positive(func):
    def wrapper(number):
        if number <= 0:
            raise ValueError("عدد باید بزرگتر از صفر باشد!")
        return func(number)
    return wrapper

@validate_positive
def square(number):
    return number ** 2

# تست تابع
print(square(5))  # خروجی: 25
print(square(-3)) # خطا: ValueError
```

در اینجا، `validate_positive` decorator بررسی می‌کند که ورودی تابع `square` مثبت باشد. اگر ورودی منفی باشد، خطای `ValueError` ایجاد می‌کند.

ب. لاگ‌کردن (Logging)

می‌توانیم از decorator برای لاگ‌کردن اطلاعات مختلف در طول اجرای توابع استفاده کنیم، مثل زمان اجرای تابع یا مقادیر ورودی و خروجی آن.

```
import time

def log_function(func):
    def wrapper(*args, **kwargs):
        print(f"فراخوانی {func.__name__} با ورودی‌ها: {args} و {kwargs}")
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"{func.__name__} اجرا شد و زمان: {end_time - start_time} ثانیه طول کشید")
        return result
    return wrapper

@log_function
def add(a, b):
    return a + b

# تست تابع
add(3, 5)
```

خروجی:

```
فراخوانی add با ورودی‌ها: (3, 5) {}
اجرا شد و زمان: 0.00002 ثانیه طول کشید add
```

در اینجا، `log_function` تابع `add` را تغییر می‌دهد و اطلاعات مربوط به ورودی‌ها و زمان اجرای آن را چاپ می‌کند.

---

3. کاربرد Decorator برای مجوزها (Permissions)

در برخی مواقع ممکن است بخواهیم از decorators برای مدیریت دسترسی به توابع استفاده کنیم. مثلاً می‌توانیم از یک decorator برای اطمینان از داشتن مجوزهای کافی برای انجام یک عمل خاص استفاده کنیم.

```
def requires_permission(permission):
    def decorator(func):
        def wrapper(*args, **kwargs):
            if permission != "admin":
                raise PermissionError("شما مجوز دسترسی ندارید!")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@requires_permission("admin")
def delete_user():
    print("کاربر حذف شد!")

# تست تابع
delete_user()  # موفقیت‌آمیز

@requires_permission("guest")
def delete_user():
    print("کاربر حذف شد!")

delete_user()  # خطا: PermissionError
```

در اینجا، `requires_permission` decorator بررسی می‌کند که آیا کاربر دارای مجوز "admin" است یا نه. اگر مجوز کافی نباشد، دسترسی به تابع مسدود می‌شود.

---

4. استفاده از `functools.wraps`

هنگام استفاده از decoratorها، ویژگی‌های تابع اصلی مانند نام (`__name__`)، مستندات (`__doc__`) و دیگر ویژگی‌ها ممکن است تغییر کند. برای حفظ این ویژگی‌ها می‌توان از `functools.wraps` استفاده کرد.

```
from functools import wraps

def my_decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print("قبل از اجرای تابع")
        return func(*args, **kwargs)
    return wrapper
```

```
@my_decorator
def greet(name):
    """این تابع برای خوشامدگویی به نام افراد است."""
    print(f"سلام، {name}!")

print(greet.__name__)  # خروجی: greet
print(greet.__doc__)  # خروجی: این تابع برای خوشامدگویی به نام افراد است.
```

در اینجا، استفاده از `@wraps(func)` اطمینان می‌دهد که تابع `greet` نام و مستندات اصلی خود را حفظ کند.

نتیجه‌گیری

Decorators ابزاری قدرتمند برای افزودن ویژگی‌های اضافی به توابع و کلاس‌ها در پایتون هستند. از این ویژگی می‌توان برای اعتبارسنجی ورودی‌ها، لاگ‌کردن، مدیریت دسترسی و بسیاری از موارد دیگر استفاده کرد. همچنین، با استفاده از `functools.wraps` می‌توان ویژگی‌های تابع اصلی را حفظ کرد.

مزایای استفاده از Decorators:

- ساده و تمیز کردن کد.
- اضافه کردن ویژگی‌های جدید بدون تغییر در کد اصلی.
- افزایش خوانایی و نگهداری آسان‌تر کد.