Decorators in Python

English Explanation

Introduction

A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure. Decorators are a powerful and useful tool in Python since they allow programmers to modify the behavior of a function or class. Decorators allow us to wrap another function in order to extend the behavior of the wrapped function, without permanently modifying it.

Example 1: Calculating Execution Time

```
import time

# Decorator function to calculate execution time
def calculate_execution_time(func):
    def wrapper_function(*args, **kwargs):
        print('Start execution...')
        start_time = time.time()  # Record start time
        func()  # Execute the original function
        end_time = time.time()  # Record end time
        print(f'Execution time: {end_time - start_time}')  # Print execution time
        print('End execution...')

    return wrapper_function

# Function to be decorated
@calculate_execution_time
def myfunc():
    time.sleep(5)  # Simulate a delay

myfunc()  # Call the decorated function
```

Explanation:

- `calculate_execution_time`: This is a decorator function that calculates the time taken to execute a function.
- `wrapper_function`: This inner function wraps the original function (`func`) to add the functionality of calculating and printing the execution time.
- `@calculate_execution_time`: This syntax is used to apply the decorator to `myfunc`.
- When `myfunc` is called, it sleeps for 5 seconds, and the decorator calculates and prints the execution time.

Example 2: Adding Timestamps

```
import datetime

# Decorator function to add timestamps
def decorator(func):
    def wrapper_function(*args, **kwargs):
        print('Start...')
        func(*args, **kwargs)  # Execute the original function
        print(datetime.datetime.now())  # Print the current date and time
        print('End decorator')

    return wrapper_function

# Function to be decorated
@decorator
def sum(x, y):
    print(x + y)  # Print the sum of two numbers

sum(4, 7)  # Call the decorated function
```

Explanation:

- `decorator`: This is a decorator function that prints the start message, calls the original function, and then prints the current date and time.
- `wrapper_function`: This inner function wraps the original function (`func`) to add the start message and timestamp functionality.
- `@decorator`: This syntax is used to apply the decorator to the `sum` function.
- When `sum` is called with the arguments 4 and 7, it prints the sum (11), the current date and time, and the end message.

Example 3: Using a Decorator Class

```
from datetime import datetime

# Decorator class to add functionality
class DecoratorClass:
    def __init__(self, func):
        self.func = func  # Store the original function

    def __call__(self, *args, **kwargs):
        print('Start...')
        print(self.func(3, 4))  # Call the original function with arguments 3 and
4
        print(datetime.now())  # Print the current date and time

# Function to be decorated
@DecoratorClass
def sum(x, y):
    return x + y  # Return the sum of two numbers

sum(4, 7)  # Call the decorated function
```

Explanation:

- `DecoratorClass`: This is a class that acts as a decorator. The `__init__` method initializes the class with the function to be decorated, and the `__call__` method allows the class instance to be called as a function.
- `@DecoratorClass`: This syntax is used to apply the decorator class to the `sum` function.
- When `sum` is called with the arguments 4 and 7, it prints the start message, the sum of 3 and 4 (as specified in the decorator), and the current date and time.

Common Examples of Decorators

- `login_required`
- `permission_required`
- `cache`

شرح باللغة العربية

مقدمة

المزخرف (Decorator) هو نمط تصميم في بايثون يسمح للمستخدم بإضافة وظيفة جديدة إلى كائن موجود دون تعديل هيكله. المزخرفات أدوات قوية ومفيدة في بايثون لأنها تسمح للمبرمجين بتعديل سلوك الدالة أو الفئة. المزخرفات تسمح لنا بتغليف دالة أخرى لإضافة سلوك إضافي لها دون تعديلها بشكل دائم.

المثال 1: حساب وقت التنفيذ

```
import time

# دالة المزخرف لحساب وقت التنفيذ
def calculate_execution_time(func):
    def wrapper_function(*args, **kwargs):
        print('بدء التنفيذ...')
        start_time = time.time()  # تسجيل وقت البدء
        func()  # تنفيذ الدالة الأصلية
        end_time = time.time()  # تسجيل وقت النهاية
        print(f'وقت التنفيذ: {end_time - start_time}')  # طباعة وقت التنفيذ
        print('نهاية التنفيذ...')

    return wrapper_function

# دالة لتزيينها
@calculate_execution_time
def myfunc():
    time.sleep(5)  # محاكاة تأخير

myfunc()  # استدعاء الدالة المزينة
```

الشرح:

- `calculate_execution_time`: هذه دالة مزخرف تحسب الوقت المستغرق لتنفيذ دالة.
- `wrapper_function`: هذه الدالة الداخلية تغلف الدالة الأصلية (`func`) لإضافة وظيفة حساب وعرض وقت التنفيذ.

- `@calculate_execution_time`: هذه الصيغة تستخدم لتطبيق المزخرف على `myfunc`.
- تنام الدالة لمدة 5 ثواني، والمزخرف يحسب ويعرض وقت التنفيذ، عند استدعاء `myfunc`.

المثال 2: إضافة الطوابع الزمنية

```
import datetime

# دالة المزخرف لإضافة الطوابع الزمنية
def decorator(func):
    def wrapper_function(*args, **kwargs):
        print('بدء...')
        func(*args, **kwargs)  # تنفيذ الدالة الأصلية
        print(datetime.datetime.now())  # طباعة التاريخ والوقت الحاليين
        print('نهاية المزخرف')

    return wrapper_function

# دالة لتزيينها
@decorator
def sum(x, y):
    print(x + y)  # طباعة مجموع العددين

sum(4, 7)  # استدعاء الدالة المزينة
```

الشرح:

- `decorator`: هذه دالة مزخرف تطبع رسالة البدء، تستدعي الدالة الأصلية، ثم تطبع التاريخ والوقت الحاليين.
- `wrapper_function`: هذه الدالة الداخلية تغلف الدالة الأصلية (`func`) لإضافة رسالة البدء ووظيفة الطابع الزمني.
- `@decorator`: هذه الصيغة تستخدم لتطبيق المزخرف على دالة `sum`.
- عند استدعاء `sum` مع الوسيطين 4 و 7، تطبع مجموع الأعداد (11)، والتاريخ والوقت الحاليين، ورسالة النهاية.

المثال 3: استخدام فئة المزخرف

```
from datetime import datetime

# فئة المزخرف لإضافة وظائف
class DecoratorClass:
    def __init__(self, func):
        self.func = func  # تخزين الدالة الأصلية

    def __call__(self, *args, **kwargs):
        print('بدء...')
        print(self.func(3, 4))  # استدعاء الدالة الأصلية مع الوسائط 3 و 4
        print(datetime.now())  # طباعة التاريخ والوقت الحاليين

# دالة لتزيينها
@DecoratorClass
def sum(x, y):
    return x + y  # إرجاع مجموع العددين

sum(4, 7)  # استدعاء الدالة المزينة
```

الشرح:

- `DecoratorClass`: هذه فئة تعمل كمزخرف. دالة `__init__` تقوم بتهيئة الفئة بالدالة المراد زخرفتها، ودالة `__call__` تسمح لاستدعاء مثيل الفئة كدالة.
- `@DecoratorClass`: هذه الصيغة تستخدم لتطبيق فئة المزخرف على دالة `sum`.
- عند استدعاء `sum` مع الوسيطين 4 و 7، تطبع رسالة البدء، ومجموع الأعداد 3 و 4 (كما هو محدد في المزخرف)، والتاريخ والوقت الحاليين.

أمثلة شائعة للمزخرفات

- `login_required`
- `permission_required`
- `cache`

This document provides a comprehensive guide to understanding and using decorators in Python, with explanations and examples in both English and Arabic.