TEMA IV

DESARROLLO DE APLICACIONES DE PROPÓSITO GENERAL

"Un sueño no se hace realidad por arte de magia, necesita sudor, determinación y trabajo duro"
Colin Powell

4. 2 DECORADORES

"A través de los decoradores seremos capaces reducir las líneas de código duplicadas, haremos que nuestro código sea legible, fácil de testear, fácil de mantener y, sobre todo, tendremos un código mucho más "Pythonico"."

Un decorador es una función que toma como input una función y a su vez retorna otra función. Al implementar un decorador estaremos trabajando, con por lo menos, 3 funciones: El input, el output y la función principal. El objetivo es alterar el funcionamiento original de la función que se pasa como parámetro.

Al nosotros utilizar la palabra **decorar** estamos indicando que queremos modificar el comportamiento de una función ya existente, pero sin tener que modificar su código. Esto es muy útil, principalmente, cuando queremos extender nuevas funcionalidades a dicha función. De allí el nombre **decorar**.

Hay funciones que tienen en común muchas funcionalidades, por ejemplo, las de manejo de errores de conexión de recursos I/O (que se deben programar siempre que usemos estos recursos) o las de validación de permisos en las respuestas de peticiones de servidores, en vez de repetir el código de rutinas podemos abstraer, bien sea el manejo de error o la respuesta de peticiones, en una función decorador.

Veamos unos ejemplos sencillos:

1. Queremos modificar el comportamiento de la función que suma dos números pasados por parámetro para que en todos los casos al resultado final le sume 2.

```
def mi_suma(suma):
    def nueva_suma(a,b):
        return a+b+2
    return nueva_suma


@mi_suma
def suma(x,y):
    return x+y


print(suma(2,4))
```

2. Sólo añadimos 2 si la suma resultante es mayor de 10

3. Al ejemplo anterior le pasamos como parámetro la cantidad a añadir

```
def mi_suma(arg):
    def wrapper_mi_suma(suma):
        def nueva_suma(a,b):
            sum=a+b
            if (sum>10):
                return sum+arg
            else:
                return sum
        return nueva_suma
    return wrapper_mi_suma


@mi_suma(5)
def suma(x,y):
    return x+y


print(suma(2,14))
```

Hay ejemplos reales muy útiles del uso de decoradores

1. Autorización

Los decoradores permiten verificar si alguien está o no autorizado a usar una determinada función, por ejemplo, en una aplicación web. Son muy usados en *frameworks* como Flask o Django.

```
from functools import wraps

def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not check_auth(auth.username, auth.password):
            authenticate()
        return f(*args, **kwargs)
    return decorated
```

2. Iniciar sesión

```
from functools import wraps

def logit(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print (func.__name__ + " was called")
        return func(*args, **kwargs)
    return with_logging

@logit
def addition_func(x):
   """Función suma"""
   return x + x


result = addition_func(4)
# Salida: addition_func was called
```

4.3 GENERADORES

El Generador de datos permitirá un flujo de datos **(o Streaming)** en tiempo real y tener cada uno de los datos solo cuando necesite ser consumido

La iteración es uno de los más comunes patrones de programación en Python. Los programas hacen mucha iteración para procesar listas, leer archivos, consultar una base de datos, y más. Una de las características más poderosas de Python es la habilidad de costumizar y redefinir la iteración en una función generadora.

Un generador es una función que retorna los valores con la instrucción yield y no return.

```
def generador():
   yield 5
```

Una función generadora se diferencia de una función normal en que tras ejecutar el `yield`, la función devuelve el control a quién la llamó, pero la función es pausada y el estado (valor de las variables) es guardado. Esto permite que su ejecución pueda ser reanudada más adelante.

Iterando los Generadores

Otra de las características que hacen a los *generators* diferentes, es que pueden ser iterados, ya que codifican los métodos `__iter__()` y `__next__()`, por lo que podemos usar `next()` sobre ellos. Dado que son iterables, lanzan también un `StopIteration` cuando se ha llegado al final.

Volviendo al ejemplo anterior, vamos a ver cómo podemos usar el `next()`.

```
a = generador()
print(next(a))
# Salida: 5

a = generador()
print(next(a))
print(next(a))
# Salida: 5
# Salida: Error! StopIteration:
```

tenemos una excepción del tipo `StopIteration`, ya que el generador no devuelve más valores. Esto se debe a que cada vez que usamos `next()` sobre el generador, se llama y se continúa su ejecución después del último `yield`. Y en este caso cómo no hay más código, no se generan más valores.

Creando Generadores

Vamos a ver otro ejemplo donde tengamos un generador que genere varios valores. En la siguiente función podemos ver como tenemos una variable n que, incrementada en 1, y después retorna con `yield`. Lo que pasará aquí, es que el generador generará un total de tres valores.

```
def generador():
    n = 1
    yield n

    n += 1
    yield n

    n += 1
    yield n
```

Y haciendo uso de `next()` al igual que hacíamos antes, podemos ver los valores que han sido generados. Lo que pasa por debajo, sería lo siguiente:

- Se entra en la función generadora, n=1 y se devuelve ese valor. Como ya hemos visto, el estado de la función se guarda (el valor de `n` es guardado para la siguiente llamada)
- La segunda vez que usamos `next()` se entra otra vez en la función, pero se continúa su ejecución donde se dejó anteriormente. Se suma 1 a la `n` y se devuelve el nuevo valor.
- La tercera llamada, realiza lo mismo.
- Una cuarta llamada daría un error, ya que no hay más código que ejecutar.

```
g = generador()
print(next(g))
print(next(g))
print(next(g))
# Salida: 1, 2, 3
```

Otra forma más cómoda de realizar lo mismo sería usando un simple bucle for, ya que el generador es iterable.

```
for i in generador():
    print(i)
# Salida: 1, 2, 3
```

Ya hemos mencionado que el uso de los generadores hace que no todos los valores estén almacenados en memoria, sino que sean generados al vuelo.

Ejemplo: "Calcular la suma de los n primeros números". Si n es un número muy muy grande podríamos tener problemas de memoria, usando generadores esto no nos pasará