Un Análisis Exhaustivo de los Decoradores en Python: Desde los Principios Funcionales hasta la Implementación Avanzada

Sección 1: Fundamentos Funcionales de los Decoradores en Python

Para comprender en su totalidad el concepto de los decoradores en Python, es imperativo primero explorar los principios de programación funcional sobre los que se asientan. Los decoradores no son una característica mágica o aislada del lenguaje; por el contrario, son una aplicación elegante y una consecuencia directa de las capacidades funcionales de Python, que permiten un alto grado de metaprogramación. Esta sección establece la base teórica esencial, demostrando que la existencia de los decoradores es el resultado lógico de tratar a las funciones como ciudadanos de primera clase, la capacidad de crear funciones de orden superior y el poderoso mecanismo de las clausuras (closures).

1.1 El Paradigma de las Funciones como Objetos de Primera Clase

En el núcleo del diseño de Python se encuentra el principio de que casi todo es un objeto, y las funciones no son una excepción. En Python, las funciones son consideradas "objetos de primera clase" (first-class objects).[1] Este término significa que una función puede ser tratada de la misma manera que cualquier otro objeto de datos, como un entero, una cadena de texto o una lista. Esta característica es el pilar fundamental que habilita la existencia y la sintaxis idiomática de los decoradores.[1]

Las implicaciones de que las funciones sean objetos de primera clase son profundas y se manifiestan en varias capacidades clave [3]:

1. **Asignación a variables:** Una función puede ser asignada a una variable. La variable se convierte entonces en una referencia a esa función y puede ser utilizada para invocarla.
2. **Almacenamiento en estructuras de datos:** Las funciones pueden ser almacenadas dentro de estructuras de datos como listas, tuplas o diccionarios.
3. **Paso como argumentos a otras funciones:** Una función puede ser pasada como un

argumento a otra función.
4. **Retorno desde otras funciones:** Una función puede ser el valor de retorno de otra función.

El siguiente ejemplo de código ilustra la primera de estas capacidades, la asignación de una función a una variable:

Python

```
def saludar(nombre):
    """Imprime un saludo personalizado."""
    print(f"Hola, {nombre}!")

# Asignar la función 'saludar' a una nueva variable 'bienvenida'
bienvenida = saludar

# Ahora 'bienvenida' es una referencia a la función 'saludar'
# y puede ser invocada como tal.
print(type(bienvenida))
bienvenida("Mundo")

# Salida:
# <class 'function'>
# Hola, Mundo!
```

La capacidad de manipular funciones como si fueran datos ordinarios no es una mera curiosidad sintáctica; es el prerrequisito esencial para la metaprogramación funcional en Python. La metaprogramación es la escritura de programas que escriben o manipulan otros programas (o a sí mismos) como sus datos. En el contexto de los decoradores, esto significa escribir funciones que manipulan otras funciones. Para que un decorador d pueda modificar el comportamiento de una función f, es indispensable que d pueda aceptar f como un argumento de entrada. Del mismo modo, para que el decorador pueda reemplazar la función original con una versión modificada, debe ser capaz de devolver una nueva función. Por lo tanto, la capacidad de pasar y devolver funciones es la piedra angular que sostiene toda la estructura de los decoradores en Python.[2]

1.2 Funciones de Orden Superior: Funciones que Operan sobre Funciones

Una consecuencia directa de tratar a las funciones como objetos de primera clase es la capacidad de crear funciones de orden superior (Higher-Order Functions, HOFs). Una

función de orden superior se define como una función que cumple al menos una de las siguientes condiciones:

- Toma una o más funciones como argumentos.
- Devuelve una función como resultado.[2]

Los decoradores son, por su propia naturaleza, una implementación del patrón de funciones de orden superior.[2] Toman una función como argumento y devuelven una nueva función, generalmente "envuelta" o "decorada" con funcionalidad adicional. El módulo functools de la biblioteca estándar de Python está dedicado a proporcionar herramientas para trabajar con funciones de orden superior, lo que subraya su importancia en el lenguaje.[5]
Para ilustrar este concepto, consideremos una función de orden superior que toma una función g y devuelve una nueva función que aplica g dos veces a su entrada:

Python

```
def aplicar_dos_veces(g):
    """
    Toma una función 'g' y devuelve una nueva función
    que aplica 'g' dos veces.
    """
    def h(y):
        # La función 'h' aplica 'g' a 'y', y luego aplica 'g' de nuevo al resultado.
        return g(g(y))

    # Devuelve la función anidada 'h'
    return h

def sumar_cinco(x):
    return x + 5

# 'aplicar_dos_veces' es una función de orden superior.
# Toma 'sumar_cinco' como argumento y devuelve una nueva función.
sumar_diez = aplicar_dos_veces(sumar_cinco)

# La nueva función 'sumar_diez' ahora aplica 'sumar_cinco' dos veces.
resultado = sumar_diez(10)
print(resultado)

# Salida:
# 20
```

En este ejemplo, aplicar_dos_veces es una función de orden superior. Acepta la función

sumar_cinco y devuelve la función h. Esta nueva función h (ahora referenciada por sumar_diez) encapsula la lógica de aplicar la función original dos veces. Este patrón es exactamente el mismo que subyace a los decoradores: tomar una función, envolverla en otra que añade comportamiento y devolver la envoltura.[4]

1.3 Clausuras (Closures): La Memoria de las Funciones Anidadas

El último pilar conceptual para entender los decoradores es el de las clausuras, o *closures*. Una clausura es una función anidada que "recuerda" y tiene acceso a las variables del ámbito de la función que la contiene (su ámbito léxico o de definición), incluso después de que la función externa haya completado su ejecución.[1]
Este mecanismo de "memoria" es fundamental para el funcionamiento de los decoradores. Cuando un decorador define una función wrapper interna, esta wrapper forma una clausura. La clausura "cierra sobre" (closes over) las variables del ámbito del decorador, la más importante de las cuales es la propia función que está siendo decorada (func).[1]
El siguiente ejemplo ilustra una clausura en acción:

Python

```
def decorador_simple(func):
    """Un decorador simple que imprime mensajes."""
    mensaje = "Función original:"

    # 'wrapper' es una función anidada que forma una clausura.
    def wrapper():
        # 'wrapper' tiene acceso a 'func' y 'mensaje' del ámbito de 'decorador_simple',
        # incluso después de que 'decorador_simple' haya retornado.
        print("Antes de llamar a la función.")
        print(f"{mensaje}")
        func()
        print("Después de llamar a la función.")

    return wrapper

def saludar():
    print("¡Hola, mundo!")

# Aplicamos el decorador explícitamente.
# 'decorador_simple' se ejecuta y devuelve la función 'wrapper'.
saludar_decorado = decorador_simple(saludar)
```

```
# En este punto, 'decorador_simple' ha terminado su ejecución.
# Sin embargo, 'saludar_decorado' (que es 'wrapper') todavía recuerda 'func' (que es 'saludar')
# y 'mensaje'.
saludar_decorado()

# Salida:
# Antes de llamar a la función.
# Función original:
# ¡Hola, mundo!
# Después de llamar a la función.
```

Las clausuras son el mecanismo que dota a los decoradores de su poder dinámico y contextual. No solo permiten que el wrapper acceda a la función original, sino que también son la base para patrones más avanzados. Por ejemplo, en una "fábrica de decoradores" (un decorador que acepta argumentos), se utiliza una doble anidación de funciones. La función más externa captura los argumentos de la fábrica en una clausura. Esta devuelve el decorador real (la función del medio), que a su vez crea otra clausura que captura la función a decorar. De este modo, la clausura actúa como un mecanismo de memoria que permite que la información (tanto la función original como los argumentos del decorador) persista desde el momento de la definición de la función hasta el momento de su ejecución.

Sección 2: Anatomía de un Decorador de Python

Con los fundamentos funcionales establecidos, ahora es posible diseccionar la estructura y sintaxis de un decorador en Python. Esta sección explora el patrón de implementación canónico, desmitifica la sintaxis @ y subraya la importancia crítica de preservar los metadatos de la función original para asegurar la robustez y la capacidad de introspección del código.

2.1 Construcción de un Decorador Básico: El Patrón de la Función Envoltorio (wrapper)

La estructura fundamental de un decorador de función en Python sigue un patrón consistente y reutilizable. Consiste en una función externa, el decorador propiamente dicho, que acepta una única función como argumento. Dentro de esta función externa, se define una segunda función, comúnmente denominada wrapper (envoltorio) o inner (interna). Esta función wrapper es la que contendrá la nueva funcionalidad y, crucialmente, es la responsable de ejecutar la función original que fue pasada al decorador. Finalmente, la función decoradora devuelve la función wrapper sin ejecutarla.[1]

La función wrapper actúa como un sustituto de la función original. Cuando el código cliente invoca lo que cree que es la función original, en realidad está invocando la función wrapper. Esta puede ejecutar código antes de la llamada a la función original, después de la llamada, o incluso decidir no llamar a la función original en absoluto, alterando así su comportamiento de forma dinámica.[6]
A continuación se muestra la anatomía de un decorador básico que añade funcionalidad antes y después de la ejecución de la función decorada:

Python

```
def mi_decorador(func):
    # 'mi_decorador' es la función externa que acepta una función 'func'.

    def wrapper():
        # 'wrapper' es la función interna que será devuelta.
        # 1. Lógica que se ejecuta ANTES de la función original.
        print("Algo está sucediendo antes de que la función sea llamada.")

        # 2. Llamada a la función original.
        func()

        # 3. Lógica que se ejecuta DESPUÉS de la función original.
        print("Algo está sucediendo después de que la función ha sido llamada.")

    # 4. El decorador devuelve la función wrapper.
    return wrapper

def di_hola():
    print("¡Hola!")

# Aplicación explícita del decorador
di_hola_decorado = mi_decorador(di_hola)
di_hola_decorado()

# Salida:
# Algo está sucediendo antes de que la función sea llamada.
# ¡Hola!
# Algo está sucediendo después de que la función ha sido llamada.
```

2.2 El Azúcar Sintáctico: Desmitificando el Símbolo @

Aunque es posible aplicar decoradores manualmente como se mostró en el ejemplo anterior (di_hola_decorado = mi_decorador(di_hola)), Python ofrece una sintaxis mucho más limpia y declarativa para este propósito: el símbolo @.[2] Esta sintaxis es un ejemplo de "azúcar sintáctico", lo que significa que es una forma más conveniente de escribir un código que podría expresarse de una manera más verbosa pero funcionalmente idéntica.[7]
La sintaxis @ se coloca en la línea inmediatamente anterior a la definición de la función que se desea decorar.[8] La siguiente declaración:

Python

```
@mi_decorador
def di_hola():
    print("¡Hola!")
```

es exactamente equivalente a:

Python

```
def di_hola():
    print("¡Hola!")

di_hola = mi_decorador(di_hola)
```

Esta equivalencia es el concepto clave para entender cómo funcionan los decoradores. En el momento de la definición, Python toma la función di_hola, la pasa como argumento a mi_decorador, y luego reasigna el nombre di_hola al objeto que mi_decorador devuelve (que es la función wrapper).[7] A partir de ese momento, cualquier llamada a di_hola() en el código invocará en realidad a la función wrapper.
La introducción de esta sintaxis, propuesta en el PEP 318, fue un paso crucial para la adopción generalizada de los decoradores, ya que hace que la intención del programador sea visualmente explícita y mantiene la lógica de decoración separada pero directamente asociada con la definición de la función.[8]

2.3 La Importancia de la Introspección: Preservando Metadatos con @functools.wraps

Un efecto secundario sutil pero problemático de la implementación básica de decoradores es la pérdida de los metadatos de la función original. Cuando una función es decorada, su

identidad es reemplazada por la del wrapper. Esto significa que atributos importantes para la introspección y la depuración, como el nombre de la función (__name__), su cadena de documentación (__doc__), su lista de argumentos y sus anotaciones de tipo (__annotations__), se pierden y son reemplazados por los del wrapper.[4]
Este comportamiento puede hacer que la depuración sea confusa, ya que los tracebacks de errores apuntarán al nombre wrapper en lugar del nombre de la función original. Además, herramientas automáticas de generación de documentación no podrán encontrar el docstring original.[10]
Para ilustrar el problema:

Python

```
def mi_decorador_simple(func):
    def wrapper():
        """Un docstring para el wrapper."""
        return func()
    return wrapper

@mi_decorador_simple
def mi_funcion_con_metadatos():
    """Un docstring para la función original."""
    pass

print(f"Nombre de la función: {mi_funcion_con_metadatos.__name__}")
print(f"Docstring: {mi_funcion_con_metadatos.__doc__}")

# Salida:
# Nombre de la función: wrapper
# Docstring: Un docstring para el wrapper.
```

Como se puede observar, la identidad de mi_funcion_con_metadatos ha sido completamente sobrescrita.
La solución idiomática y estándar para este problema es utilizar el decorador @functools.wraps de la biblioteca estándar de Python.[10] Este decorador se aplica a la función wrapper dentro de nuestro propio decorador. Su propósito es copiar los atributos relevantes de la función original (func) a la función wrapper.[11]
El uso correcto es el siguiente:

Python

```
import functools

def mi_decorador_mejorado(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        """Un docstring para el wrapper, que no será visible."""
        print("Ejecutando lógica del decorador...")
        return func(*args, **kwargs)
    return wrapper

@mi_decorador_mejorado
def mi_funcion_con_metadatos_preservados():
    """Un docstring para la función original que ahora se preserva."""
    pass

print(f"Nombre de la función: {mi_funcion_con_metadatos_preservados.__name__}")
print(f"Docstring: {mi_funcion_con_metadatos_preservados.__doc__}")

# Salida:
# Nombre de la función: mi_funcion_con_metadatos_preservados
# Docstring: Un docstring para la función original que ahora se preserva.
```

Además de preservar los metadatos, @functools.wraps añade un atributo __wrapped__ a la función wrapper, que contiene una referencia a la función original sin decorar. Esto es extremadamente útil para la introspección avanzada, ya que permite "desenvolver" el decorador y acceder a la función subyacente si es necesario.[11]
La utilización de @functools.wraps trasciende la mera "buena práctica" para convertirse en un requisito esencial para la creación de decoradores robustos y profesionales. En ecosistemas de software complejos, herramientas como depuradores, generadores de documentación (por ejemplo, Sphinx), frameworks de pruebas y otras utilidades de introspección dependen de metadatos precisos para su correcto funcionamiento. Un decorador que no preserva estos metadatos rompe la funcionalidad de estas herramientas, creando un componente que no se integra limpiamente en el ecosistema de Python. Por tanto, @functools.wraps es el mecanismo que garantiza la transparencia del decorador no solo a nivel de comportamiento, sino también a nivel de metadatos, lo cual es fundamental para la mantenibilidad, la depuración y la integración del software.

Sección 3: Patrones de Decoradores Avanzados

Una vez dominada la estructura básica de los decoradores, es posible extender su funcionalidad para crear herramientas de metaprogramación más potentes y flexibles. Esta

sección explora patrones avanzados que permiten a los decoradores manejar cualquier tipo de función, aceptar sus propios argumentos de configuración y ser combinados para construir comportamientos complejos de manera modular y declarativa.

3.1 Creación de Decoradores Universales con *args y **kwargs

Los decoradores básicos presentados anteriormente tienen una limitación significativa: solo pueden decorar funciones que no aceptan argumentos. Si se intenta aplicar un decorador como mi_decorador a una función que requiere parámetros, se producirá un TypeError en tiempo de ejecución, ya que la función wrapper interna se definió sin parámetros y, por lo tanto, no puede aceptar los argumentos destinados a la función original.
Para construir un decorador de propósito general, o universal, que pueda aplicarse a cualquier función independientemente de su firma (es decir, del número y tipo de sus argumentos), la función wrapper debe ser capaz de aceptar un número arbitrario de argumentos posicionales y de palabra clave. Esto se logra en Python utilizando las sintaxis especiales *args y **kwargs.[3]

- *args: Recopila todos los argumentos posicionales pasados a la función en una tupla.
- **kwargs (keyword arguments): Recopila todos los argumentos de palabra clave en un diccionario.[12]

Al definir la función wrapper con la firma def wrapper(*args, **kwargs), esta puede aceptar cualquier combinación de argumentos. Posteriormente, estos argumentos deben ser pasados a la función original func al invocarla, utilizando la misma sintaxis de "desempaquetado": func(*args, **kwargs). Esto asegura que la función original reciba exactamente los mismos argumentos con los que fue llamada la función decorada.[7]
El siguiente ejemplo muestra un decorador de temporización universal:

Python

```
import functools
import time

def temporizador(func):
    """Decorador que mide y muestra el tiempo de ejecución de una función."""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        inicio = time.perf_counter()
        # Pasa todos los argumentos posicionales y de palabra clave a la función original
        resultado = func(*args, **kwargs)
        fin = time.perf_counter()
        duracion = fin - inicio
```

```
        print(f"La función '{func.__name__}' tardó {duracion:.4f} segundos en ejecutarse.")
        return resultado
    return wrapper

@temporizador
def calculo_complejo(n, factor=2):
    """Realiza un cálculo que consume tiempo."""
    total = 0
    for i in range(n):
        total += i * factor
    return total

# La función decorada puede ser llamada con diferentes argumentos
resultado1 = calculo_complejo(10000000)
resultado2 = calculo_complejo(20000000, factor=3)
```

Este decorador temporizador ahora puede aplicarse a cualquier función, sin importar cuántos argumentos posicionales o de palabra clave reciba, lo que lo convierte en una herramienta de perfilado reutilizable y no intrusiva.

3.2 Fábricas de Decoradores: Decoradores que Aceptan Argumentos

A menudo, es deseable que un decorador sea configurable. Por ejemplo, un decorador que reintenta una operación podría necesitar que se le especifique el número de reintentos, o un decorador de logging podría necesitar un nivel de registro específico. Para lograr esto, se utiliza un patrón conocido como "fábrica de decoradores".[4]
Una fábrica de decoradores es una función que acepta argumentos y *devuelve* un decorador. Esto introduce una capa adicional de anidación en la estructura de la función. La estructura de tres niveles es la siguiente [3]:

1. **Función Fábrica (Externa):** Acepta los parámetros de configuración del decorador (p. ej., veces=3). Su única responsabilidad es devolver la función decoradora.
2. **Función Decoradora (Intermedia):** Es la función que será devuelta por la fábrica. Acepta la función a decorar (func) como su único argumento. Su responsabilidad es devolver la función wrapper.
3. **Función Envoltorio (wrapper):** Es la función más interna. Acepta los argumentos de la función decorada (*args, **kwargs). Contiene la lógica principal del decorador y utiliza los parámetros que fueron capturados por la clausura de la función fábrica.

La magia de las clausuras es lo que permite que este patrón funcione. La función wrapper tiene acceso no solo a func (del ámbito de la función decoradora), sino también a los parámetros originales pasados a la función fábrica.
El siguiente ejemplo implementa un decorador repetir que acepta un argumento veces para

especificar cuántas veces debe ejecutarse la función decorada [13]:

Python

```
import functools

def repetir(veces):
    """Fábrica de decoradores que repite la ejecución de una función."""
    # 1. Función Fábrica: acepta el argumento 'veces'.
    def decorador(func):
        # 2. Función Decoradora: acepta la función 'func'.
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # 3. Función Wrapper: tiene acceso a 'veces' y 'func'.
            resultado_final = None
            for _ in range(veces):
                resultado_final = func(*args, **kwargs)
            return resultado_final
        return wrapper
    return decorador

# La sintaxis @ ahora incluye argumentos para la fábrica.
@repetir(veces=3)
def saludar(nombre):
    print(f"¡Hola, {nombre}!")

saludar("Equipo")

# Salida:
# ¡Hola, Equipo!
# ¡Hola, Equipo!
# ¡Hola, Equipo!
```

Cuando Python encuentra @repetir(veces=3), primero llama a repetir(veces=3). Esta llamada devuelve la función decorador. Luego, Python utiliza este decorador devuelto para decorar la función saludar, lo que equivale a saludar = decorador(saludar).

3.3 El Arte de Apilar Decoradores: Composición y Orden de Ejecución

Python permite aplicar múltiples decoradores a una única función, un proceso conocido

como "apilamiento" o "encadenamiento" de decoradores.[1] La sintaxis es simple: se colocan los decoradores uno encima del otro.

Python

```
@decorador1
@decorador2
def mi_funcion():
    pass
```

Comprender el orden en que se aplican estos decoradores es fundamental. Los decoradores se aplican desde el más cercano a la función hacia afuera, es decir, de abajo hacia arriba.[4] La expresión anterior es sintácticamente equivalente a la siguiente composición de funciones:

Python

```
mi_funcion = decorador1(decorador2(mi_funcion))
```

Esto significa que decorador2 envuelve primero a mi_funcion, y luego decorador1 envuelve el resultado de esa primera decoración. En consecuencia, en tiempo de ejecución, el wrapper de decorador1 se ejecutará primero, y desde dentro de él se llamará al wrapper de decorador2, que a su vez llamará a la función original mi_funcion.
El siguiente ejemplo aclara este orden de ejecución:

Python

```
import functools

def decorador_uno(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("Entrando en el decorador UNO")
        resultado = func(*args, **kwargs)
        print("Saliendo del decorador UNO")
        return resultado
    return wrapper

def decorador_dos(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
```

```
        print("  Entrando en el decorador DOS")
        resultado = func(*args, **kwargs)
        print("  Saliendo del decorador DOS")
        return resultado
    return wrapper

@decorador_uno
@decorador_dos
def decir_algo():
    print("    ¡Dentro de la función original!")

decir_algo()

# Salida:
# Entrando en el decorador UNO
#   Entrando en el decorador DOS
#     ¡Dentro de la función original!
#   Saliendo del decorador DOS
# Saliendo del decorador UNO
```

El resultado muestra un patrón de "anidamiento": el código del decorador más externo (decorador_uno) envuelve completamente la ejecución del decorador más interno (decorador_dos), que a su vez envuelve a la función original. Este comportamiento es predecible y permite componer funcionalidades complejas a partir de decoradores más simples y enfocados, promoviendo un código limpio y modular.

Sección 4: Decoradores en el Contexto de la Programación Orientada a Objetos

Si bien los decoradores tienen sus raíces en la programación funcional, su utilidad se extiende de manera poderosa al paradigma de la programación orientada a objetos (POO) en Python. Se pueden utilizar para modificar el comportamiento de métodos individuales dentro de una clase o incluso para alterar o registrar clases enteras. Esta sección explora la implementación de decoradores utilizando clases, la creación de decoradores con estado, las particularidades de decorar los diferentes tipos de métodos de una clase y el uso de los decoradores incorporados que son fundamentales en la POO de Python.

4.1 Decoradores Basados en Clases: Implementación con __init__ y __call__

Además de utilizar funciones para crear decoradores, Python permite que las propias clases actúen como decoradores. Este enfoque puede ofrecer una mayor flexibilidad y una mejor organización del código, especialmente cuando el decorador necesita mantener un estado o su lógica es compleja.[14]
Para que una clase funcione como un decorador, debe ser un objeto "callable" (invocable). Esto se logra implementando el método especial __call__. El proceso de decoración con una clase sigue un flujo lógico [9]:

1. **Inicialización (__init__):** Cuando se decora una función (por ejemplo, con @MiDecoradorDeClase), Python crea una instancia de MiDecoradorDeClase. La función que se está decorando se pasa como argumento al constructor __init__ de la clase. Típicamente, el constructor almacena esta función en un atributo de la instancia (por ejemplo, self.func).
2. **Invocación (__call__):** Cuando se llama a la función decorada en el código, Python invoca el método __call__ de la instancia del decorador que la reemplazó. Este método contiene la lógica del decorador, similar a la función wrapper en un decorador funcional. Desde __call__, se puede ejecutar la función original almacenada.

El siguiente ejemplo reimplementa el decorador temporizador utilizando una clase:

Python

```
import functools
import time

class Temporizador:
    def __init__(self, func):
        # El constructor recibe la función a decorar y la almacena.
        functools.update_wrapper(self, func)
        self.func = func

    def __call__(self, *args, **kwargs):
        # __call__ se ejecuta cuando se invoca la instancia.
        # Contiene la lógica del decorador.
        inicio = time.perf_counter()
        resultado = self.func(*args, **kwargs)
        fin = time.perf_counter()
        duracion = fin - inicio
        print(f"La función '{self.func.__name__}' tardó {duracion:.4f} segundos en ejecutarse.")
        return resultado

@Temporizador
def calculo_largo(n):
```

```
    """Suma números en un bucle."""
    return sum(i for i in range(n))

calculo_largo(10000000)

# Salida:
# La función 'calculo_largo' tardó 0.3813 segundos en ejecutarse.
```

Nótese que en lugar de @functools.wraps, se utiliza functools.update_wrapper(self, func) para copiar los metadatos de la función original a la instancia del decorador, logrando el mismo efecto de preservación de la introspección.

4.2 El Poder del Estado: Creación de Decoradores Stateful

La principal ventaja de los decoradores basados en clases sobre sus contrapartes funcionales es la facilidad con la que pueden mantener un estado a lo largo de múltiples llamadas. Mientras que un decorador de función requeriría el uso de variables nonlocal o el almacenamiento del estado como un atributo de la propia función wrapper (lo cual puede ser menos intuitivo), una clase puede utilizar sus atributos de instancia (self.atributo) de forma natural y explícita para este propósito.[1]
Un decorador con estado (stateful) es aquel que recuerda información entre invocaciones. Un caso de uso clásico es un decorador que cuenta cuántas veces se ha llamado a una función.

Python

```
import functools

class ContadorDeLlamadas:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        # Inicializa el estado (el contador) en el constructor.
        self.num_llamadas = 0

    def __call__(self, *args, **kwargs):
        # Cada vez que se llama, se actualiza el estado.
        self.num_llamadas += 1
        print(f"Llamada número {self.num_llamadas} a la función '{self.func.__name__}'")
        return self.func(*args, **kwargs)

@ContadorDeLlamadas
```

```
def saludar():
    print("¡Hola de nuevo!")

saludar()
saludar()
saludar()

# Salida:
# Llamada número 1 a la función 'saludar'
# ¡Hola de nuevo!
# Llamada número 2 a la función 'saludar'
# ¡Hola de nuevo!
# Llamada número 3 a la función 'saludar'
# ¡Hola de nuevo!
```

En este ejemplo, self.num_llamadas es un atributo de la instancia de ContadorDeLlamadas que envuelve a saludar. Este atributo persiste entre las llamadas, permitiendo al decorador mantener un registro de su historial de invocaciones de una manera limpia y encapsulada.[7]

4.3 Decorando Métodos de Instancia: El Manejo de self y el Protocolo Descriptor

Decorar métodos de instancia de una clase introduce una capa adicional de complejidad. Un método de instancia se diferencia de una función regular en que su primer argumento es siempre una referencia a la instancia de la clase, convencionalmente llamada self. Un decorador aplicado a un método de instancia debe manejar correctamente este argumento self.[17]
Cuando se utiliza un decorador de función, la solución es sencilla: la función wrapper debe aceptar self como su primer argumento y pasarlo a la función original.

Python

```
def log_metodo(func):
    @functools.wraps(func)
    def wrapper(self, *args, **kwargs): # Acepta 'self' como primer argumento
        print(f"Llamando al método '{func.__name__}' en la instancia {self}")
        return func(self, *args, **kwargs) # Pasa 'self' a la función original
    return wrapper

class MiClase:
```

```
    def __init__(self, valor):
        self.valor = valor

    @log_metodo
    def mi_metodo(self, incremento):
        self.valor += incremento
        print(f"Nuevo valor: {self.valor}")

instancia = MiClase(10)
instancia.mi_metodo(5)

# Salida:
# Llamando al método 'mi_metodo' en la instancia <__main__.MiClase object at...>
# Nuevo valor: 15
```

Sin embargo, cuando se utiliza un decorador de clase, surge un problema sutil. La sintaxis instancia.mi_metodo() no es una simple llamada a función. Python primero busca el atributo mi_metodo en instancia. Si este atributo es una función definida en la clase, Python invoca el "protocolo de descriptor" para crear un "método enlazado" (bound method), que es un objeto que empaqueta la función y la instancia instancia. Es este método enlazado el que luego se llama.
Un decorador de clase simple interrumpe este mecanismo. Al decorar mi_metodo, el atributo instancia.mi_metodo ya no apunta a una función, sino a la instancia del decorador. Cuando se llama, se invoca decorador_instancia.__call__(). El problema es que el self dentro de __call__ es la propia instancia del decorador, no la instancia de MiClase.
La solución idiomática en Python es convertir la clase decoradora en un "descriptor" implementando el método especial __get__. Un descriptor es un objeto que tiene un comportamiento personalizado cuando se accede a él como atributo de otro objeto. El método __get__ se invoca cuando se accede al atributo. Permite al decorador "capturar" la instancia de la clase que se está utilizando (instancia) y devolver un objeto enlazado que se puede llamar correctamente.[19]

Python

```
import functools

class DecoradorDeMetodo:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, owner):
        # __get__ se llama cuando se accede al método.
```

```
        # 'instance' es la instancia de la clase cliente (o None si se accede desde la clase).
        if instance is None:
            return self
        # Devuelve una nueva función (usando functools.partial) que ya tiene 'instance'
        # enlazado como el primer argumento ('self').
        return functools.partial(self.__call__, instance)

    def __call__(self, instance, *args, **kwargs):
        # Ahora __call__ recibe explícitamente la instancia.
        print(f"Decorador de clase ejecutándose para el método '{self.func.__name__}' en
{instance}")
        return self.func(instance, *args, **kwargs)

class OtraClase:
    @DecoradorDeMetodo
    def metodo_decorado(self):
        print("Ejecutando el método original.")

obj = OtraClase()
obj.metodo_decorado()

# Salida:
# Decorador de clase ejecutándose para el método 'metodo_decorado' en
<__main__.OtraClase object at...>
# Ejecutando el método original.
```

Este patrón, aunque más complejo, es la forma robusta y correcta de crear decoradores de clase que funcionen de manera transparente con métodos de instancia.

4.4 Análisis de los Decoradores Incorporados: @classmethod, @staticmethod y @property

Python proporciona varios decoradores incorporados que son fundamentales para la programación orientada a objetos. Los más importantes son @classmethod, @staticmethod y @property.[20]

- **@staticmethod**: Transforma un método para que no reciba un primer argumento implícito (ni self ni cls). Se comporta como una función regular que está lógicamente agrupada dentro de la clase (en su espacio de nombres). Es útil para funciones de utilidad que no dependen del estado de la clase ni de la instancia.[18]
- **@classmethod**: Transforma un método para que reciba la clase misma como primer argumento implícito, convencionalmente llamado cls. Puede acceder y modificar el

estado de la clase (atributos de clase), pero no el estado de una instancia específica. Un caso de uso común es la creación de "fábricas de métodos" o constructores alternativos.[18]

- **@property**: Transforma un método en un atributo de "solo lectura". Permite llamar a un método sin usar paréntesis, como si se accediera a un atributo. Esto es útil para calcular valores derivados del estado de un objeto. El decorador @property también puede ser utilizado para definir *setters* y *deleters* para el atributo, permitiendo un control fino sobre cómo se accede, se modifica y se elimina un atributo.[20]

La siguiente tabla resume las diferencias clave entre los métodos de instancia, de clase y estáticos, sirviendo como una guía de referencia rápida para su uso correcto.[18]

Tipo de Método	Decorador	Primer Argumento	Acceso al Estado de Instancia	Acceso al Estado de Clase	Caso de Uso Principal
Método de Instancia	Ninguno	self (la instancia)	Sí (lectura/escritura)	Sí (vía self.__class__)	Operaciones que dependen del estado de un objeto específico.
Método de Clase	@classmethod	cls (la clase)	No	Sí (lectura/escritura)	Fábricas de métodos, constructores alternativos, operaciones a nivel de clase.
Método Estático	@staticmethod	Ninguno	No	No	Funciones de utilidad lógicamente relacionadas con la clase.

La elección correcta entre estos tipos de métodos es crucial para un diseño de clases claro y mantenible. Un método de instancia debe usarse cuando la lógica depende del estado único de un objeto. Un método de clase es apropiado cuando la operación se relaciona con la clase en su conjunto, como crear una instancia a partir de una representación alternativa (por ejemplo, datetime.fromtimestamp()). Un método estático es la elección correcta cuando una función está conceptualmente ligada a una clase pero no necesita acceder ni a la clase ni a sus instancias.

4.5 Decoradores para Clases Completas: Modificación Colectiva del Comportamiento

Finalmente, es importante distinguir entre usar una clase *como* un decorador y aplicar un decorador *a* una clase. Un decorador de clase es una función que toma una clase como argumento y devuelve una clase nueva o modificada.[24]
Este patrón es menos común que los decoradores de funciones, pero es extremadamente poderoso para realizar modificaciones a nivel de toda la clase. El ejemplo más prominente en la biblioteca estándar es @dataclasses.dataclass, que automáticamente añade a una clase métodos especiales como __init__, __repr__, __eq__, entre otros, basándose en las anotaciones de tipo de sus atributos.[24]
Otro caso de uso es la implementación de patrones de diseño como el Singleton, o para registrar automáticamente una clase en un registro central. El siguiente ejemplo muestra un decorador simple que añade un nuevo método a una clase:

Python

```
def anadir_metodo_saludo(cls):
    """Decorador de clase que añade un método 'saludar'."""
    def saludo_generico(self):
        print(f"Un saludo desde una instancia de {cls.__name__}")

    # Añade el nuevo método a la clase
    setattr(cls, 'saludar', saludo_generico)
    return cls

@anadir_metodo_saludo
class MiClaseSimple:
    pass

instancia = MiClaseSimple()
instancia.saludar()

# Salida:
# Un saludo desde una instancia de MiClaseSimple
```

Los decoradores de clase, aunque conceptualmente similares a los decoradores de función (ambos son funciones de orden superior), operan a un nivel de abstracción más alto, permitiendo la manipulación programática de la estructura y comportamiento de las clases en el momento de su definición.[4]

Sección 5: Aplicaciones Prácticas y Casos de Uso en el

Mundo Real

La verdadera utilidad de los decoradores se manifiesta en su capacidad para resolver problemas comunes de desarrollo de software de una manera elegante, reutilizable y no intrusiva. Al separar las "preocupaciones transversales" (cross-cutting concerns) como el logging, el caching o la autorización de la lógica de negocio principal, los decoradores promueven un código más limpio y mantenible, adhiriéndose al principio de "No te repitas" (Don't Repeat Yourself, DRY).[1] Esta sección explora varios casos de uso prácticos que demuestran el poder de los decoradores en escenarios del mundo real.

5.1 Implementación de Sistemas de Registro (Logging) y Medición de Rendimiento

Una de las aplicaciones más comunes y directas de los decoradores es añadir capacidades de registro (logging) y temporización a las funciones. Sin decoradores, un programador tendría que añadir manualmente llamadas de logging o código de temporización al principio y al final de cada función que desee monitorear, lo cual es repetitivo y propenso a errores.[2]
Un decorador permite encapsular esta lógica una sola vez y aplicarla de forma declarativa donde sea necesario.
Ejemplo de Decorador de Logging:

Python

```
import functools
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

def logger(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        logging.info(f"Ejecutando '{func.__name__}' con args: {args}, kwargs: {kwargs}")
        try:
            resultado = func(*args, **kwargs)
            logging.info(f"'{func.__name__}' finalizó exitosamente. Resultado: {resultado}")
            return resultado
        except Exception as e:
            logging.error(f"Excepción en '{func.__name__}': {e}")
```

```
        raise # Relanza la excepción para no alterar el flujo de errores
    return wrapper

@logger
def sumar(a, b):
    return a + b

sumar(5, 3)
```

Este decorador logger registra la entrada y la salida de la función sumar, así como cualquier excepción que pueda ocurrir, proporcionando una traza de auditoría completa sin modificar la simple lógica return a + b.[17]

5.2 Optimización mediante Caching y Memoización: @functools.lru_cache

El caching o la memoización es una técnica de optimización que consiste en almacenar los resultados de llamadas a funciones costosas y devolver el resultado almacenado en caché cuando se producen las mismas entradas nuevamente.[2] Los decoradores con estado, especialmente los basados en clases, son perfectos para implementar esta funcionalidad. Python, reconociendo la utilidad de este patrón, proporciona un decorador de memoización altamente optimizado y listo para usar en su biblioteca estándar: @functools.lru_cache.[5] Este decorador implementa una estrategia de caché "Least Recently Used" (LRU), que descarta los elementos menos utilizados recientemente cuando la caché alcanza su tamaño máximo.
Parámetros y Uso:

- maxsize: El número máximo de resultados a almacenar. Si se establece en None, la caché puede crecer indefinidamente. El valor predeterminado es 128.
- typed: Si es True, los argumentos de diferentes tipos se almacenarán en caché por separado (por ejemplo, f(3) y f(3.0) se tratarán como llamadas distintas).

El decorador también proporciona métodos para la introspección, como cache_info() para ver las estadísticas de aciertos y fallos, y cache_clear() para invalidar la caché.[5]
Ejemplo con la Sucesión de Fibonacci:
El cálculo recursivo de la sucesión de Fibonacci es un ejemplo clásico de una función que se beneficia enormemente de la memoización, ya que implica muchos cálculos redundantes.

Python

```
import functools
import time
```

```
# Sin caché
def fibonacci_sin_cache(n):
    if n < 2:
        return n
    return fibonacci_sin_cache(n-1) + fibonacci_sin_cache(n-2)

# Con caché LRU
@functools.lru_cache(maxsize=None)
def fibonacci_con_cache(n):
    if n < 2:
        return n
    return fibonacci_con_cache(n-1) + fibonacci_con_cache(n-2)

# Medición del rendimiento
start_time = time.time()
fibonacci_sin_cache(35)
print(f"Sin caché, tiempo: {time.time() - start_time:.4f} segundos")

start_time = time.time()
fibonacci_con_cache(35)
print(f"Con caché, tiempo: {time.time() - start_time:.4f} segundos")

print(fibonacci_con_cache.cache_info())

# Salida:
# Sin caché, tiempo: 2.1458 segundos
# Con caché, tiempo: 0.0000 segundos
# CacheInfo(hits=33, misses=36, maxsize=None, currsize=36)
```

La diferencia de rendimiento es drástica. El decorador @lru_cache transforma una función con complejidad exponencial en una con complejidad lineal, simplemente añadiendo una línea de código.

5.3 Mecanismos de Autorización y Control de Acceso

En muchas aplicaciones, especialmente en sistemas web y APIs, es necesario restringir el acceso a ciertas funciones basándose en los permisos del usuario. Los decoradores ofrecen una forma declarativa y limpia de implementar esta lógica de autorización.[1]
Un decorador de autorización puede verificar si un usuario tiene el rol o los permisos necesarios antes de ejecutar la función protegida. Si el usuario no está autorizado, el decorador puede lanzar una excepción o redirigirlo, evitando que el código de la función se

ejecute.[17]
Este patrón es utilizado extensivamente en frameworks web como Flask y Django.[1]
Ejemplo de Decorador de Autorización:

Python

```
import functools

# Simulación de un usuario actual
current_user = {'username': 'admin', 'role': 'admin'}
# current_user = {'username': 'guest', 'role': 'guest'}

def requiere_rol(rol):
    """Fábrica de decoradores que verifica el rol del usuario."""
    def decorador(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            if current_user.get('role') == rol:
                return func(*args, **kwargs)
            else:
                raise PermissionError(f"Acceso denegado. Se requiere el rol '{rol}'.")
        return wrapper
    return decorador

@requiere_rol('admin')
def panel_de_administracion():
    print("Bienvenido al panel de administración.")

@requiere_rol('guest')
def pagina_de_invitado():
    print("Bienvenido, invitado.")

panel_de_administracion()
# pagina_de_invitado() # Lanzaría PermissionError si el usuario es 'admin'
```

5.4 Validación de Argumentos y Precondiciones

Los decoradores son una excelente herramienta para la programación por contrato, permitiendo validar que los argumentos de una función cumplen ciertas precondiciones antes

de que se ejecute la lógica principal. Esto separa la lógica de validación de la lógica de negocio, haciendo ambas más limpias y fáciles de mantener.[6]
Un decorador puede, por ejemplo, verificar los tipos de los argumentos, asegurarse de que los valores numéricos estén dentro de un rango válido o que las cadenas de texto no estén vacías.
Ejemplo de Decorador de Validación de Tipos:

Python

```
import functools

def validar_tipos(*tipos_esperados):
    """Fábrica de decoradores que valida los tipos de los argumentos posicionales."""
    def decorador(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for arg, tipo_esperado in zip(args, tipos_esperados):
                if not isinstance(arg, tipo_esperado):
                    raise TypeError(f"Argumento '{arg}' no es del tipo esperado
{tipo_esperado.__name__}")
            return func(*args, **kwargs)
        return wrapper
    return decorador

@validar_tipos(int, int)
def dividir(a, b):
    return a / b

print(dividir(10, 2))
# dividir(10, "2") # Lanzaría TypeError
```

Este decorador validar_tipos asegura la integridad de los datos de entrada de la función dividir de una manera declarativa y reutilizable.[3]

5.5 El Rol de los Decoradores en Frameworks Web Modernos

Los decoradores son una pieza central en la arquitectura de muchos frameworks web modernos, como Flask y Django. Su capacidad para asociar funciones con eventos o configuraciones de manera concisa los hace ideales para tareas como el enrutamiento de URLs.

En Flask, por ejemplo, el decorador @app.route() se utiliza para vincular una URL a la función que debe manejar las solicitudes a esa URL. Este es un ejemplo paradigmático de una fábrica de decoradores que no solo modifica la función, sino que también la registra en una estructura de datos interna del framework para un uso posterior.[4]
Ejemplo de Enrutamiento en Flask:

Python

```
# from flask import Flask # Se necesitaría instalar Flask para ejecutar esto

# app = Flask(__name__)

# @app.route('/')
# def index():
#     return "Página de inicio"

# @app.route('/usuario/<username>')
# def perfil_usuario(username):
#     return f"Página del usuario {username}"
```

En este caso, @app.route('/') es una llamada a un método que actúa como una fábrica de decoradores. Toma la ruta de la URL como argumento y devuelve un decorador que registra la función index para que sea llamada cuando se reciba una solicitud HTTP para la ruta /. Este uso de decoradores hace que el código del framework sea extremadamente legible y fácil de usar, ocultando la complejidad del mecanismo de enrutamiento subyacente.

Sección 6: Conclusiones y Recomendaciones de Diseño

Los decoradores en Python representan una de las características más potentes y elegantes del lenguaje, encarnando la flexibilidad que ofrece su modelo de datos y sus capacidades de programación funcional. A lo largo de este análisis, se ha demostrado que los decoradores son mucho más que un simple "azúcar sintáctico"; son una herramienta de metaprogramación fundamental que permite a los desarrolladores extender y modificar el comportamiento de funciones y clases de manera limpia, modular y reutilizable.

6.1 Síntesis de la Versatilidad de los Decoradores

La versatilidad de los decoradores se deriva de su capacidad para inyectar lógica antes, después o alrededor de la ejecución de un objeto callable. Esto permite abordar una amplia gama de "preocupaciones transversales" sin alterar el código original, lo que conduce a un software de mayor calidad. Los beneficios clave identificados incluyen:

- **Aumento de la Funcionalidad:** Los decoradores permiten añadir dinámicamente nuevas responsabilidades a los objetos, como logging, caching, control de acceso y validación, sin necesidad de recurrir a la herencia o a la modificación directa del código fuente.[6]
- **Promoción de la Reutilización de Código:** La lógica común, como la medición del rendimiento o la gestión de reintentos, puede encapsularse en un decorador y aplicarse a múltiples funciones en todo un proyecto, promoviendo el principio DRY (Don't Repeat Yourself).[1]
- **Mejora de la Legibilidad y Mantenibilidad:** La sintaxis @ proporciona una forma declarativa y visualmente clara de indicar que una función tiene un comportamiento modificado. Esto separa la lógica de negocio principal de las preocupaciones secundarias, haciendo que el código sea más fácil de leer, entender y mantener a largo plazo.[6]

6.2 Directrices para el Diseño: Cuándo Usar Decoradores de Función vs. de Clase

Una decisión de diseño clave al implementar un decorador es si utilizar una función o una clase. Aunque a menudo son intercambiables, cada enfoque tiene fortalezas que lo hacen más adecuado para ciertos escenarios.[15]

- **Use un decorador de función cuando:**
    - **La lógica es simple y sin estado:** Para decoradores sencillos que no necesitan recordar información entre llamadas (por ejemplo, un logger simple o un temporizador), un decorador basado en una función es generalmente más conciso y fácil de leer.
    - **La familiaridad es una prioridad:** El patrón de función anidada wrapper es el más comúnmente encontrado y entendido por la mayoría de los desarrolladores de Python.
- **Use un decorador de clase cuando:**
    - **Se necesita mantener un estado (stateful):** Esta es la razón principal para elegir un decorador de clase. Los atributos de la instancia (self.atributo) proporcionan una forma natural y limpia de almacenar estado entre llamadas, como en contadores, cachés o sistemas que necesitan recordar un estado anterior.[1]
    - **La lógica del decorador es compleja:** Si la lógica del decorador es intrincada y se beneficia de ser descompuesta en múltiples métodos, una clase ofrece una mejor estructura y encapsulación que una función compleja con múltiples

funciones anidadas.
    - **Se desea aprovechar la herencia:** Se pueden crear jerarquías de decoradores, donde una clase base de decorador define un comportamiento común y las subclases lo extienden o modifican. Esto no es posible con decoradores de función.

6.3 Principios para la Creación de Decoradores Robustos, Reutilizables y Mantenibles

Para concluir, la creación de decoradores de alta calidad que se integren sin problemas en un ecosistema de software más amplio requiere la adhesión a un conjunto de mejores prácticas.

1. **Preservar siempre los metadatos:** Utilice @functools.wraps en la función wrapper (o functools.update_wrapper en el __init__ de un decorador de clase). Este es el principio más importante para garantizar que los decoradores no interfieran con la depuración, la documentación y otras herramientas de introspección.
2. **Diseñar para la generalidad:** A menos que un decorador esté diseñado para un caso de uso muy específico, su función wrapper (o método __call__) debe aceptar *args y **kwargs y pasarlos a la función original. Esto asegura que el decorador sea aplicable a la más amplia gama posible de funciones.
3. **Documentar claramente el comportamiento:** El docstring del decorador debe explicar claramente qué hace, qué argumentos (si los hay) acepta y cómo modifica el comportamiento de la función decorada.
4. **Gestionar el estado con cuidado:** Si se crea un decorador con estado, sea consciente de las implicaciones. El estado es global para todas las llamadas a esa instancia de función decorada, lo que puede tener efectos secundarios inesperados, especialmente en entornos concurrentes. Los decoradores de clase hacen que la gestión del estado sea más explícita.
5. **Considerar el protocolo de descriptor para métodos:** Al crear decoradores de clase destinados a ser utilizados en métodos de instancia, es fundamental implementar el método __get__ para que el decorador se comporte correctamente como un descriptor y maneje adecuadamente la instancia self.

En resumen, los decoradores son una manifestación de la elegancia y el poder del diseño de Python. Cuando se comprenden sus fundamentos funcionales y se aplican con disciplina y buenas prácticas de diseño, se convierten en una herramienta indispensable en el arsenal de cualquier desarrollador de Python para escribir código más expresivo, modular y mantenible.

Fuentes citadas

1. How to Use Python Decorators (With Function and Class-Based Examples) - DataCamp, acceso: agosto 16, 2025, https://www.datacamp.com/tutorial/decorators-python
2. What Is a Decorator in Python? | Coursera, acceso: agosto 16, 2025,

https://www.coursera.org/articles/what-is-a-decorator-in-python
3. Decorators with parameters in Python - GeeksforGeeks, acceso: agosto 16, 2025, https://www.geeksforgeeks.org/python/decorators-with-parameters-in-python/
4. Higher-Order Functions and Decorators | by Andrei Lapets | Python Supply - Medium, acceso: agosto 16, 2025, https://medium.com/python-supply/higher-order-functions-and-decorators-d6bb31a5c78d
5. functools — Higher-order functions and operations on callable ..., acceso: agosto 16, 2025, https://docs.python.org/3/library/functools.html
6. The Python Decorator Handbook - freeCodeCamp, acceso: agosto 16, 2025, https://www.freecodecamp.org/news/the-python-decorator-handbook/
7. 12.2. Python Decorators — Fundamentals of Web Programming - Runestone Academy, acceso: agosto 16, 2025, https://runestone.academy/ns/books/published/webfundamentals/Frameworks/decorators.html
8. PythonDecorators - Python Wiki, acceso: agosto 16, 2025, https://wiki.python.org/moin/PythonDecorators
9. Python Decorators - __call__ in class - Stack Overflow, acceso: agosto 16, 2025, https://stackoverflow.com/questions/19497771/python-decorators-call-in-class
10. Python Decorator: Why You Should Use functools.wraps | by Seunghyun Yoo - Medium, acceso: agosto 16, 2025, https://medium.com/@blueberry92450/using-functools-wraps-in-python-decorator-952030a70615
11. A Deep Dive Into Python's functools.wraps Decorator - Jacob Padilla, acceso: agosto 16, 2025, https://jacobpadilla.com/articles/functools-deep-dive
12. What are args** and kwargs** and __somethinghere__ in python? - Reddit, acceso: agosto 16, 2025, https://www.reddit.com/r/learnpython/comments/1g48ci8/what_are_args_and_kwargs_and_somethinghere_in/
13. Decorators With Parameters In Python - Flexiple, acceso: agosto 16, 2025, https://flexiple.com/python/decorators-parameters-python
14. Class as Decorator in Python - GeeksforGeeks, acceso: agosto 16, 2025, https://www.geeksforgeeks.org/python/class-as-decorator-in-python/
15. Class decorators vs function decorators - python - Stack Overflow, acceso: agosto 16, 2025, https://stackoverflow.com/questions/6676015/class-decorators-vs-function-decorators
16. Python Class Decorators: A Guide | Built In, acceso: agosto 16, 2025, https://builtin.com/software-engineering-perspectives/python-class-decorator
17. How To Use Self With Decorator? - GeeksforGeeks, acceso: agosto 16, 2025, https://www.geeksforgeeks.org/python/how-to-use-self-with-decorator/
18. Python's Instance, Class, and Static Methods Demystified – Real ..., acceso: agosto 16, 2025, https://realpython.com/instance-class-and-static-methods-demystified/
19. Decorating bound-methods in Python — a general and scalable solution -

Sebastian Ahmed, acceso: agosto 16, 2025, https://sebastian-ahmed.medium.com/decorating-bound-methods-in-python-a-general-and-scalable-solution-b16579c3a469
20. Python: Decorators in OOP. A guide on classmethods, staticmethods... | by Aniruddha Karajgi | TDS Archive | Medium, acceso: agosto 16, 2025, https://medium.com/data-science/python-decorators-in-oop-3189c526ead6
21. Class method vs Static method in Python - GeeksforGeeks, acceso: agosto 16, 2025, https://www.geeksforgeeks.org/python/class-method-vs-static-method-python/
22. What is the difference between @staticmethod and @classmethod in Python?, acceso: agosto 16, 2025, https://stackoverflow.com/questions/136097/what-is-the-difference-between-staticmethod-and-classmethod-in-python
23. Built-in Functions — Python 3.13.7 documentation, acceso: agosto 16, 2025, https://docs.python.org/3/library/functions.html
24. Deep Dive into Python Class Decorator | by E.Y. - Medium, acceso: agosto 16, 2025, https://elfi-y.medium.com/deep-dive-into-python-class-decorator-1a22e8d7845f