- Seminario 5: Python Mágico
  - Resolución de miembros y métodos en Python
  - Funcionamiento de super()
  - Métodos mágicos
    - Métodos mágicos de Python más comunes
  - Métodos built-in de Python
    - Iteradores
      - Método eval()
  - Tema fuera del Seminario: Decoradores

Seminario 5: Python Mágico

Problema a implementar

Ver notebook

Solución propuesta en el problema del seminario 5:

```
class Matriz:
    def __init__(self, filas, columnas, valor_inicial=0):
        self.filas = filas
        self.columnas = columnas
        self.matriz = [[valor_inicial]*columnas for _ in range(filas)]

    def __getitem__(self, indices):
        fila, columna = indices
        return self.matriz[fila][columna]

    def __setitem__(self, indices, valor):
        fila, columna = indices
        self.matriz[fila][columna] = valor

    def __getattr__(self, attr):
        fila, columna = map(int, attr.split('_')[1:])
        return self.matriz[fila][columna]

    def __setattr__(self, attr, valor):
        if attr.startswith('_'):
            fila, columna = map(int, attr.split('_')[1:])
            self.matriz[fila][columna] = valor
        else:
            object.__setattr__(self, attr, valor)

    def __iter__(self):
        for fila in self.matriz:
            for valor in fila:
                yield valor
```

```
    def as_type(self, tipo):
        nueva_matriz = Matriz(self.filas, self.columnas)
        nueva_matriz.matriz = [[tipo(valor) for valor in fila] for fila in
self.matriz]
        return nueva_matriz

    def __str__(self):
        return '\n'.join([' '.join(map(str, fila)) for fila in self.matriz])

    def __add__(self, other):
        if self.filas != other.filas or self.columnas != other.columnas:
            raise ValueError('Las matrices deben tener las mismas dimensiones')
        nueva_matriz = Matriz(self.filas, self.columnas)
        nueva_matriz.matriz = [[self[fila, columna] + other[fila, columna] for
columna in range(self.columnas)] for fila in range(self.filas)]
        return nueva_matriz

    def __sub__(self, other):
        if self.filas != other.filas or self.columnas != other.columnas:
            raise ValueError('Las matrices deben tener las mismas dimensiones')
        nueva_matriz = Matriz(self.filas, self.columnas)
        nueva_matriz.matriz = [[self[fila, columna] - other[fila, columna] for
columna in range(self.columnas)] for fila in range(self.filas)]
        return nueva_matriz

    def __mul__(self, other):
        if self.columnas != other.filas:
            raise ValueError('El número de columnas de la primera matriz debe ser
igual al número de filas de la segunda matriz')
        nueva_matriz = Matriz(self.filas, other.columnas)
        nueva_matriz.matriz = [[sum(self[fila, k] * other[k, columna] for k in
range(self.columnas)) for columna in range(other.columnas)] for fila in
range(self.filas)]
        return nueva_matriz
```

1. Implemente la clase `Matriz`, para representar matrices con las operaciones de suma y producto. Implemente además otras funcionalidades que crea necesarias

```
    matrizA = Matriz(2, 2, 1)
    matrizB = Matriz(2, 2, 2)

    print(f"MatrizA: \n{matrizA}\n")
    # MatrizA:
    # 1 1
    # 1 1

    print(f"MatrizB: \n{matrizB}\n")
    # MatrizB:
    # 2 2
    # 2 2

    print(f"A+B: \n{matrizA + matrizB}\n")
    # A+B:
```

```
# 3 3
# 3 3

print(f"A-B: \n{matrizA - matrizB}\n")
# A-B:
# -1 -1
# -1 -1

print(f"A*B:\n{matrizA * matrizB}\n")
# A*B:
# 4 4
# 4 4
```

2. Implemente la indización para la clase `Matriz` de forma tal que se puedan hacer construcciones como las siguientes: `a = matriz[0, 6]` o `matriz[1, 2] = 9`.

```
matrizA[1,1] = 10
print(f"A[1,1]: {matrizA[1,1]}")
# A[1,1]: 10

a = matrizA[0, 0]
print(f"A[0,0]: {a}")
# A[0,0]: 1
```

3. Implemente la indización para la clase `Matriz` por medio de acceso a campos de la forma: `a = matriz._0_6` o `matriz._1_2 = 9`.

```
matrizA._0_0 = 100
print(f"A[0,0]: {matrizA._0_0}")
# A[0,0]: 100

a = matrizA._0_0
print(f"variable a: {a}")
# A[0,0]: 100
```

4. Los objetos matrices deberán ser iterables. El iterador de una matriz con `n` filas y `m` columnas debe devolver los elementos en el siguiente orden: `matriz_1_1, matriz_1_2, ..., matriz_1_m, matriz_2_1, ..., matriz_n_m`

```
print(f"MatrizA: \n{matrizA}\n")
# MatrizA:
# 100 1
# 1 10

a = iter(matrizA)
for valor in a:
```

```
    print(valor)
# 100
# 1
# 1
# 10
```

5. Al tipo matriz se podrá aplicar siempre el método `as_type()` que devuelve una nueva matriz con todos los tipos convertidos al tipo `type`. Suponga que existe un constructor en `type` que convierte de cualquier tipo a type. Por ejemplo:

```
m = Matriz(2, 3) # crea una matriz de int con valor 0s.
mf = m.as_float() # mf es una matriz de 0s pero de tipo float.
```

```
m = Matriz(2, 3,5)
print(f"Matriz Original: \n{m}\n")
# Matriz Original:
# 5 5 5
# 5 5 5

mf = m.as_type(float)
print(f"Matriz de float: \n{mf}\n")
# Matriz de float:
# 5.0 5.0 5.0
# 5.0 5.0 5.0
```

Resolución de miembros y métodos en Python

La resolución de miembros en Python se refiere al proceso de determinar a qué clase pertenece un miembro (un método o un atributo) cuando se accede a él desde una instancia de una clase. Esto es particularmente relevante en el contexto de la herencia, donde una clase puede heredar de una o más clases base.

Python utiliza el concepto de "orden de resolución de métodos" (MRO, por sus siglas en inglés, Method Resolution Order) para determinar la secuencia en la que se buscan los métodos o atributos de una clase durante el tiempo de ejecución. El MRO se gestiona mediante el algoritmo C3, que es un algoritmo linealizador de clases. Este asegura que los métodos se resuelvan en un orden específico y evita ambigüedades

en situaciones de herencia múltiple. Es posible acceder a la MRO de una clase utilizando el atributo `__mro__` o la función `mro()`.

La función `super()` en Python se utiliza para acceder al método de la clase base en lugar del método de la clase actual. La resolución de miembros, en este contexto, se refiere a cómo se determina cuál es la clase base y, por lo tanto, cuál es el método al que se debe acceder.

Un ejemplo simple puede ayudar a entender mejor este concepto:

```
class A:
    def saludar(self):
        return "Hola desde A"

class B(A):
    def saludar(self):
        return "Hola desde B, " + super().saludar()

class C(A):
    def saludar(self):
        return "Hola desde C, " + super().saludar()

class D(B, C):
    pass

instancia_d = D()
print(instancia_d.saludar())

# Salida: Hola desde B, Hola desde C, Hola desde A
```

Funcionamiento de `super()`

Como habíamos dicho la función `super()` en Python se utiliza para llamar a métodos en la clase base en el orden definido por el MRO. Si la herencia es simple, `super()` llama recursivamente a los métodos de la clase padre. En el caso de la herencia múltiple, el uso de `super()` puede provocar resultados inesperados si no se conoce su funcionamiento. Veamos el la jerarquía definida en el ejemplo anterior:

```
graph TD
    A --> C
    A --> B
    B --> D
    C --> D
```

En este caso el orden definido por el MRO sería: D,B,C,A. Es decir, el llamado a `super()` buscaría los métodos de las clases en una especie de orden topológico, y no exactamente siguiendo la herencia de hijo a padre como se podría asumir (erróneamente). Por lo tanto, es mejor evitar el uso de `super()` en la herencia múltiple y llamar a los métodos de la clase padre directamente.

En el caso de los métodos mágicos, `super()` es especialmente útil para llamar a la implementación de la clase base de estos métodos desde una subclase.

Veamos un ejemplo con el método mágico `__init__`:

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

    class MiSubClase(MiClaseBase):
        def __init__(self, valor, otro_valor):
            super().__init__(valor)  # Llamando al __init__ de la clase base
            self.otro_valor = otro_valor

    # Creando una instancia de la subclase
    objeto = MiSubClase(10, 20)

    print(objeto.valor)       # Accediendo al atributo de la clase base
    print(objeto.otro_valor)  # Accediendo al atributo de la subclase

    # Salida:
    # 10
    # 20
```

Métodos mágicos

Una clase puede implementar ciertas operaciones que son invocadas por una sintaxis especial (como operaciones aritméticas o indexación) mediante la definición los **métodos especiales**. Este es el enfoque de `Python` para la sobrecarga de operadores, permitiendo que las clases definan su propio comportamiento con respecto a los operadores del lenguaje. Por ejemplo, si una clase define un método llamado `__getitem__()`, y x es una instancia de esta clase, entonces x[i] es aproximadamente equivalente a type(x).`__getitem__(x, i)`. Si no se define un método apropiado, entonces los intentos de ejecutar una operación generan excepción (normalmente AttributeError o TypeError).

Establecer un método especial en None u omitirlo, indica que la operación correspondiente no está disponible. Por ejemplo, si una clase no establece `__iter__()` , entonces la clase no es iterable, por lo que llamar a iter() en sus instancias generará un TypeError (sin recurrir a `__getitem__()`).

Por qué los métodos especiales tienen una sintaxis rara?

"...the second bit of Python rationale I promised to explain is the reason why I chose special methods to look `__special__` and not merely special. I was anticipating lots of operations that classes might want to override, some standard (e.g. `__add__` or `__getitem__`), some not so standard (e.g. pickle's `__reduce__` for a long time had no support in C code at all). I didn't want these special operations to use ordinary method names, because then pre-existing classes, or classes written by users without an encyclopedic memory for all the special methods, would be liable to accidentally define operations they didn't mean to implement, with possibly disastrous consequences. Ivan Krstić explained this more concise in his message, which arrived after I'd written all this up..." - Guido van Rossum

Métodos mágicos de Python más comunes

`__new__` y `__init__` son ambos métodos mágicos en Python que se utilizan en la creación de objetos, pero tienen algunas diferencias clave:

- `__new__` es el primer paso en la creación de un objeto. Toma la clase como primer argumento, seguido de cualquier otro argumento que se pase al constructor. Su principal responsabilidad es crear y devolver la nueva instancia del objeto. Este método es raramente sobrescrito, excepto en casos muy particulares o cuando se está manipulando aspectos de bajo nivel de la creación del objeto.
- `__init__` es el segundo paso en la creación de un objeto, después de `__new__`. Toma la nueva instancia creada por `__new__` (que se pasa como el primer argumento, usualmente denominado `self`) y cualquier otro argumento que se pase al constructor, y lo utiliza para inicializar el objeto. Este es el método que se sobrescribe más comúnmente cuando se crea una nueva clase para definir cómo se inicializan los objetos de esa clase.

En resumen, `__new__` se encarga de la creación del objeto, y `__init__` se encarga de la inicialización del objeto una vez creado.

```python
class MiClase:
    def __new__(cls, *args, **kwargs):
        print("MiClase.__new__ llamado")
        instancia = object.__new__(cls)
        return instancia

    def __init__(self, valor):
        print("MiClase.__init__ llamado")
        self.valor = valor

mi_instancia = MiClase("Hola, mundo")

# Salida:
# MiClase.__new__ llamado
# MiClase.__init__ llamado
```

`__repr__` y `__str__` son ambos métodos mágicos en Python que se utilizan para representar un objeto de una manera legible. Ambos métodos son llamados cuando se utiliza la función `print()` para imprimir un objeto, pero hay algunas diferencias clave:

- `__repr__` se utiliza para devolver la representación oficial de un objeto, si es posible debería ser una expresión válida de Python que pudiera utilizarse para recrear un objeto con el mismo valor (dado un entorno apropiado).
- `__str__` se utiliza para devolver una representación legible de un objeto. Es llamado con la función `print()` o `str()`. Si no se define `__str__`, Python llamará a `__repr__` en su lugar.

```python
class MiClase:
    def __init__(self, valor):
        self.valor = valor

    def __repr__(self):
        return f"MiClase({self.valor!r})"

    # def __str__(self):
    #     return f"Instancia de MiClase con el valor {self.valor!r}"

instancia = MiClase(10)
print("Si no se encuentra __str__, se imprime __repr__")
print(instancia)

# Salida:
# Si no se encuentra __str__, se imprime __repr__
# MiClase(10)
```

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

    def __repr__(self):
        return f"MiClase({self.valor!r})"

    def __str__(self):
        return f"Instancia de MiClase con el valor {self.valor!r}"

instancia = MiClase(10)
print("Imprimiendo con __str__")
print(instancia)
print("Imprimiendo la __repr__")
print(repr(instancia))

# Salida:
# Imprimiendo con __str__
# Instancia de MiClase con el valor 10
# Imprimiendo la __repr__
# MiClase(10)
```

Métodos de comparación y su correspondencia con los operadores de comparación

```
object.__lt__(self, other) # self < other
object.__le__(self, other) # self <= other
object.__eq__(self, other) # self == other
object.__ne__(self, other) # self != other
object.__gt__(self, other) # self > other
object.__ge__(self, other) # self >= other
```

En ocasiones no es necesario tener explícitamente todas implementadas, ya que Python puede inferir el resultado de una operación de comparación a partir de otras. Por ejemplo, si `__eq__` y `__ne__` no están implementados, Python invocará `__lt__` y `__gt__` para determinar el resultado de `__ne__`.

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

    def __lt__(self, otro):
        return self.valor < otro.valor

    def __le__(self, otro):
        return self.valor <= otro.valor

    def __gt__(self, otro):
```

```
            return self.valor > otro.valor

    a = MiClase(10)
    b = MiClase(20)

    print(a == b) # False
    print(a != b) # True
    print(a < b)  # True
    print(a <= b) # True
    print(a > b) # False
    print(a >= b) # False
```

Sobrecarga de operadores numéricos y su correspondencia con los operadores aritméticos

```
    object.__add__(self, other) # self + other
    object.__sub__(self, other) # self - other
    object.__mul__(self, other) # self * other
    object.__matmul__(self, other) # self @ other
    object.__truediv__(self, other) # self / other
    object.__floordiv__(self, other) # self // other
    object.__mod__(self, other) # self % other
    object.__divmod__(self, other) # divmod(self, other)
    object.__pow__(self, other[, modulo]) # self ** other or pow(self, other,
modulo)
    object.__lshift__(self, other) # self << other
    object.__rshift__(self, other) # self >> other
    object.__and__(self, other) # self & other
    object.__xor__(self, other) # self ^ other
    object.__or__(self, other)  # self | other
```

Ejemplo:

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

        def __add__(self, otro):
            return self.valor + otro.valor

    a = MiClase(10)
    b = MiClase(20)

    print(a + b)  # Salida: 30
```

- `__getitem__` y `__setitem__` Son llamados con la sintaxis self[key] y self[key] = value respectivamente. Como su nombre indica, se utilizan para obtener y establecer valores.

- `__len__` Debe devolver la longitud del objeto.
- `__iter__` Debe devolver un iterador para el objeto. Debe poder ser usado en un bucle for.
- `__next__` Debe devolver el siguiente elemento en el iterador.
- `__contains__` Debe devolver True si el objeto contiene el valor especificado, False en caso contrario.

```
class MiClase:
    def __init__(self, lista):
        self.lista = lista

    def __getitem__(self, indice):
        return self.lista[indice]

    def __setitem__(self, indice, valor):
        self.lista[indice] = valor

    def __len__(self):
        return len(self.lista)

    def __iter__(self):
        for valor in self.lista:
            yield valor

mi_lista = MiClase([1, 2, 3, 4, 5])

print("modificando valores de la lista:")
print(mi_lista[0])  # Salida: 1
mi_lista[0] = 10
print(mi_lista[0])  # Salida: 10

print("obteniendo la longitud de la lista:")
print(len(mi_lista))  # Salida: 5


print("iterando sobre la lista:")
for valor in mi_lista:
    print(valor)

# Salida:
# 10 2 3 4 5
```

`__call__` Define un comportamiento para la instancia, cuando es llamada como función.

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

```
        def __call__(self, otro):
            return self.valor + otro
        def __str__(self):
            return str(self.valor)

    a = MiClase(10)
    b = a(20)
    print(b)  # Salida: 30
```

Hay muchos otros métodos mágicos que se pueden implementar en una clase, no es necesario tenerlos todos implementados, pero pueden ser útiles en diferentes situaciones. La documentación oficial de Python proporciona una lista completa de los métodos mágicos disponibles y su comportamiento esperado.

Métodos built-in de Python

En Python, los métodos built-in son métodos que están disponibles en las clases por defecto, sin necesidad de importar ningún módulo adicional. A diferencia de los métodos de las clases, su sintaxis es en notación prefija, es decir `metodo(objeto, argumentos)` en lugar de `objeto.metodo(argumentos)`. Esto es solo una cuestión de estilo, explicado por el creador de Python.

La mayoría de los métodos built-in son funciones que llaman a los métodos mágicos de las clases, como `len()` que llama a `__len__`, `iter()` que llama a `__iter__`, `next()` que llama a `__next__`, entre otros. Algunos métodos built-in son funciones que realizan operaciones específicas, como `print()` que imprime un objeto en la consola, `type()` que devuelve el tipo de un objeto, `id()` que devuelve el identificador único de un objeto, entre otros.

Iteradores

Los iteradores en Python son objetos que implementan los métodos `__iter__()` y `__next__()`, lo que les permite ser iterados (recorridos) a través de un bucle `for`. Los iteradores proporcionan una forma eficiente y elegante de acceder a elementos de una secuencia, como listas, tuplas, diccionarios, conjuntos, y otros objetos iterables.

Un objeto iterador en Python debe cumplir con dos requisitos principales:

1. `__iter__()`: Este método debe devolver el propio objeto iterador. Es llamado al iniciar la iteración y se espera que devuelva un objeto iterador.
2. `__next__()`: Este método debe devolver el próximo elemento en la secuencia. Cuando no hay más elementos para iterar, se espera que levante la excepción `StopIteration`.

Veamos un ejemplo usando iteradores y métodos built-in:

```
class Contador:
    def __init__(self, limite):
        self.limite = limite
        self.valor = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.valor < self.limite:
            resultado = self.valor
            self.valor += 1
            return resultado
        else:
            raise StopIteration

# Crear un objeto iterador
contador_iterador = Contador(5)
# Utilizar el iterador en un bucle for
for numero in contador_iterador:
    print(numero)

# Salida:
# 0
# 1
# 2
# 3
# 4

class Iterable:
    _index = 0
    def __init__(self, *args):
        self.valores = args
    def __len__(self):
        return len(self.valores)
    def __iter__(self):
        for valor in self.valores:
            yield valor
    def __getitem__(self, index):
        return self.valores[index]

a = Iterable(1, 2, 3, 4, 5)
print(f"\nlenghth de a: {len(a)}")
print(list(a)) # funciona xq Iterable tiene el metodo __iter__
```

```
print(list(reversed(a))) # funciona xq Iterable tiene el metodo __getitem__
```

Método `eval()`

El método `eval()` es un método built-in que evalúa una expresión en forma de cadena y devuelve el resultado. Puede ser útil en la construcción de AST o en la evaluación de expresiones dinámicas. Por ejemplo:

```
    expr = "print('Hola, mundo!')"
    eval(expr) # Hola, mundo!

    expr  = "x**2 + 2*x + 1"
    x = 2
    print(eval(expr, {'x': x})) # 9
```

`eval()` recibe un string y lo evalúa como una expresión de Python, también puede recibir un diccionario con variables locales y globales. Esto nos puede ayudar a evaluar expresiones dinámicas o mejorar la seguridad al usar eval.

```
    safe_list = ['acos', 'asin', 'atan', 'atan2', 'ceil', 'cos',
                 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor',
                 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10',
                 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt',
                 'tan', 'tanh']

    safe_dict = {k: globals().get(k, None) for k in safe_list} # globals devuelve
un diccionario con las variables globales
    print("Imprimiendo el diccionario de funciones seguras")
    print(safe_dict)

    expr = "x**e + 2*x + 1"
    x = 2
    print("Expresion " + expr + f" con x = {x}")
    print(eval(expr, {'x': x}, safe_dict))

    # ejemplo de error
    # expr = "inverso([1,1,1,1])"
    # eval(expr, safe_dict) # NAME ERROR 'inverso' is not define

    def inverso(lista):
        return [1/x for x in lista]

    safe_dict['inverso'] = inverso

    print("\nAgregando la funcion inverso al diccionario de funciones seguras")
    expr = "inverso([1,2,3,4])"
    print(eval(expr, safe_dict))
```

```
    # Salida:
    # Imprimiendo el diccionario de funciones seguras
    # {'acos': <built-in function acos>, 'asin': <built-in function asin>, 'atan':
<built-in function atan>, 'atan2': <built-in function atan2>, ...}

    # Expresion x**e + 2*x + 1 con x = 2
    # 11.58088599101792

    # Agregando la funcion inverso al diccionario de funciones seguras
    # [1.0, 0.5, 0.3333333333333333, 0.25]
```

De igual manera existen muchos otros métodos built-in que pueden ser útiles en diferentes situaciones. La documentación oficial de Python proporciona una lista completa de los métodos built-in disponibles, y sus llamados a los metodos especiales.

Tema fuera del Seminario: Decoradores

En Python, un decorador es una función que toma otra función o método y extiende o modifica su comportamiento sin modificar su código fuente. Los decoradores son una forma elegante y poderosa de realizar acciones adicionales antes o después de la ejecución de una función. Pueden usarse para reutilizar y extender el comportamiento de funciones de manera modular.

Explicación básica de cómo funcionan los decoradores en Python:

1. **Sintaxis básica de un decorador:**

   Un decorador se aplica a una función utilizando la sintaxis `@decorador` justo encima de la definición de la función. Por ejemplo:

   ```
   @mi_decorador
   def mi_funcion():
       # código de la función
   ```

   En este caso, `mi_funcion` será pasada como argumento a la función `mi_decorador`.

2. **Definición de un decorador:**

   Un decorador es simplemente una función en Python. Puede aceptar una función como argumento, realizar alguna acción, y luego devolver una función modificada

o extender el comportamiento de la función original.

```
def mi_decorador(funcion_original):
    def funcion_modificada():
        print("Realizando acciones antes de llamar a la función original")
        funcion_original()
        print("Realizando acciones después de llamar a la función original")
    return funcion_modificada
```

3. **Aplicación de un decorador:**

   Se puede aplicar un decorador a una función utilizando la sintaxis `@`:

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

   En este caso, cuando se llama a `saludar()`, en realidad se está llamando a la versión modificada de la función creada por el decorador.

4. **Múltiples decoradores:**

   Se pueden aplicar múltiples decoradores a una función. En este caso, los decoradores se aplican de abajo a arriba:

   ```
   @decorador1
   @decorador2
   def mi_funcion():
       # código de la función
   ```

   En este ejemplo, primero se aplica `decorador2`, y luego `decorador1`.

5. **Decoradores incorporados:**

   Python proporciona algunos decoradores incorporados, como `@staticmethod`, `@classmethod`, y `@property`. Estos son utilizados comúnmente en clases para definir métodos estáticos, de clase o propiedades.

   ```
   class MiClase:
       @staticmethod
       def metodo_estatico():
           # código del método estático
   ```

```
    @classmethod
    def metodo_de_clase(cls):
        # código del método de clase

    @property
    def mi_propiedad(self):
        # código de la propiedad
```

Los decoradores son una herramienta poderosa y versátil en Python, y se utilizan ampliamente para modificar o extender el comportamiento de las funciones y métodos de una manera clara y legible.

Veamos un ejemplo sencillo y práctico de cómo se utiliza un decorador en Python. Supongamos que queremos medir el tiempo de ejecución de una función. Podemos crear un decorador que haga esto sin modificar el código interno de la función. Aquí está un ejemplo:

```
import time

# Decorador para medir el tiempo de ejecución de una función
def medir_tiempo(funcion):
    def wrapper(*args, **kwargs):
        inicio = time.time()
        resultado = funcion(*args, **kwargs)
        fin = time.time()
        print(f"{funcion.__name__} tomó {fin - inicio} segundos en
ejecutarse.")
        return resultado
    return wrapper

# Aplicar el decorador a una función
@medir_tiempo
def ejemplo_funcion():
    print("Iniciando...")
    time.sleep(2)
    print("Finalizando.")

# Llamando a la función decorada
ejemplo_funcion()

# Salida:
# Iniciando...
# Finalizando.
# ejemplo_funcion tomó 2.004222869873047 segundos en ejecutarse.
```

En este ejemplo, `medir_tiempo` es un decorador que toma una función como argumento, define una nueva función (`wrapper`) que mide el tiempo de ejecución

antes y después de llamar a la función original, e imprime el tiempo transcurrido.

Cuando aplicamos `@medir_tiempo` sobre `ejemplo_funcion`, básicamente estamos diciendo que `ejemplo_funcion` ahora se ejecutará a través de la función `wrapper` del decorador `medir_tiempo`. Cuando llamamos a `ejemplo_funcion()`, en realidad estamos llamando a `wrapper`, y el tiempo de ejecución se imprime en la consola.