1. ¿Para qué usamos Clases en Python?

Las clases en Python nos proveen una forma de empaquetar funcionalidad y datos juntos, por lo tanto se utilizan para crear objetos, que son instancias de esas clases. Cuando creamos una clase, se crea un nuevo objeto que nos permite crear nuevas instancias de ese tipo y cada una de estas instancias de clase puede tener atributos, manteniendo así su estado.

¿Por qué y para qué se utilizan?
Porque proporcionan un medio para definir las propiedades y comportamientos de los objetos. Las clases encapsulan datos (atributos) y funciones (métodos) que operan en esos datos, por lo tanto son fundamentales para la programación orientada a objetos.

Sintaxis

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Ejemplo 1:

```
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    def mostrar_info(self):
        print("Coche:", self.marca, self.modelo)

# Crear una instancia de la clase Coche
mi_coche = Coche("Toyota", "Corolla")
mi_coche.mostrar_info()

# Output:
Coche: Toyota Corolla
```

Ejemplo 2:

```
class CuentaBancaria:
    def __init__(self, titular, saldo_inicial=0):
        self.titular = titular
        self.saldo = saldo_inicial

    def depositar(self, cantidad):
```

```
        if cantidad > 0:
            self.saldo += cantidad
            print(f"Se han depositado ${cantidad}. Saldo actual: ${self.saldo}")
        else:
            print("La cantidad a depositar debe ser positiva.")

    def retirar(self, cantidad):
        if 0 < cantidad <= self.saldo:
            self.saldo -= cantidad
            print(f"Se han retirado ${cantidad}. Saldo restante: ${self.saldo}")
        else:
            print("Fondos insuficientes o cantidad inválida.")

    def mostrar_saldo(self):
        print(f"Saldo de la cuenta de {self.titular}: ${self.saldo}")

# Crear una instancia de la cuenta bancaria
cuenta_de_juan = CuentaBancaria("Juan", 1000)

# Realizar operaciones
cuenta_de_juan.depositar(500)
cuenta_de_juan.retirar(200)
cuenta_de_juan.mostrar_saldo()

# Output:
Se han depositado $500. Saldo actual: $1500
Se han retirado $200. Saldo restante: $1300
Saldo de la cuenta de Juan: $1300
```

Descripción del Código:
Definición de la Clase CuentaBancaria:

Constructor __init__: Inicializa el objeto CuentaBancaria con un titular y un saldo inicial. El saldo inicial tiene un valor por defecto de 0 si no se especifica.

Método depositar:
Acepta una cantidad a depositar. Verifica si la cantidad es positiva antes de agregarla al saldo. Imprime el estado del saldo después del depósito.

Método retirar:
Permite retirar una cantidad si esta es positiva y menor o igual al saldo disponible. Imprime el saldo restante después de la retirada.

Método mostrar_saldo:
Imprime el saldo actual de la cuenta junto con el nombre del titular.

Creación de una Instancia y Operaciones:
Se crea una instancia llamada cuenta_de_juan con Juan como titular y un saldo inicial de $1000.
Se depositan $500, se retiran $200, y finalmente se muestra el saldo actual de la cuenta.
Este ejemplo demuestra cómo las clases en Python pueden encapsular datos (como el saldo y el titular) y comportamientos (como depositar, retirar, y mostrar_saldo) relacionados de manera lógica, facilitando su manejo y mantenimiento en programas más complejos.

2. ¿Qué método se ejecuta automáticamente cuando se crea una instancia de una clase?

El método __init__ se ejecuta automáticamente cuando se crea una instancia de una clase en Python y este es el constructor de la clase.

¿Para qué se utiliza?
Se utiliza para inicializar los atributos del objeto.

Sintaxis

```
class NombreDeLaClase:
    def __init__(self, parametro1, parametro2, ...):
        self.atributo1 = parametro1
        self.atributo2 = parametro2
        # más inicializaciones de atributos si es necesario
```

Explicación:

def __init__(self, parametro1, parametro2, ...): Esto define el método __init__.
self: Es el primer parámetro que debe incluirse en cualquier método de instancia en Python. Se refiere a la instancia del objeto que se está creando.
parametro1, parametro2, ...: Estos son los parámetros que se pasan al crear una nueva instancia de la clase.
self.atributo1 = parametro1: Dentro del método __init__, se utilizan estos parámetros para inicializar los atributos de la instancia.
self.atributo2 = parametro2: Cada parámetro se asigna a un atributo específico del objeto utilizando self.

Ejemplo 1:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def mostrar_info(self):
        print("Nombre:", self.nombre)
        print("Edad:", self.edad)
```

```
# Crear una instancia de la clase Persona
persona1 = Persona("Juan", 30)
persona1.mostrar_info()

# Output
Nombre: Juan
Edad: 30
```

Ejemplo 2:

Esta vez crearemos una clase llamada Estudiante que tiene atributos para el nombre, la edad y el año de ingreso a la universidad. El método __init__ se utilizará para inicializar estos atributos al crear una nueva instancia de la clase Estudiante.

```
class Estudiante:
    def __init__(self, nombre, edad, año_ingreso):
        self.nombre = nombre
        self.edad = edad
        self.año_ingreso = año_ingreso

    def mostrar_info(self):
        print(f"Nombre: {self.nombre}")
        print(f"Edad: {self.edad} años")
        print(f"Año de ingreso: {self.año_ingreso}")

# Crear instancias de la clase Estudiante
estudiante1 = Estudiante("María", 20, 2020)
estudiante2 = Estudiante("Juan", 22, 2018)

# Mostrar información de los estudiantes
print("Información del Estudiante 1:")
estudiante1.mostrar_info()
print("\nInformación del Estudiante 2:")
estudiante2.mostrar_info()

# Output
```

Información del Estudiante 1:
Nombre: María
Edad: 20 años
Año de ingreso: 2020

Información del Estudiante 2:
Nombre: Juan
Edad: 22 años
Año de ingreso: 2018

Descripción del Código:

Definición de la Clase Estudiante:
Constructor __init__: Recibe tres parámetros (nombre, edad, año_ingreso) y los asigna como atributos del objeto.

Método mostrar_info:
Imprime la información del estudiante, incluyendo nombre, edad y año de ingreso.

Creación de Instancias y Uso:
Se crean dos instancias de la clase Estudiante: estudiante1 y estudiante2, con diferentes valores para nombre, edad y año de ingreso.

Mostrar Información de los Estudiantes:
Se llama al método mostrar_info en cada instancia para imprimir la información de cada estudiante.

3. ¿Cuáles son los tres verbos de API?

Los tres verbos principales de una API REST son:

GET: Se utiliza para obtener recursos del servidor.
POST: Se utiliza para enviar datos al servidor para crear un nuevo recurso.
DELETE: Se utiliza para eliminar un recurso en el servidor.

Estos verbos son fundamentales para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en una API.

Verbo GET
Se utiliza para obtener datos de un recurso.
En una API REST, un ejemplo común sería obtener una lista de usuarios.

Ejemplo:

Endpoint: https://api.example.com/users

Solicitud GET:

http
GET /users HTTP/1.1
Host: api.example.com

Respuesta Exitosa (200 OK):

json
```
[
  {
    "id": 1,
```

```
    "name": "John Doe",
    "email": "john@example.com"
  },
  {
    "id": 2,
    "name": "Jane Smith",
    "email": "jane@example.com"
  }
]
```

Análisis
La solicitud GET al endpoint /users devuelve una lista de usuarios en formato JSON.
La respuesta incluye los detalles de los usuarios, como su ID, nombre y correo electrónico.

Verbo POST:
Se utiliza para crear un nuevo recurso en el servidor.
En este caso, crearemos un nuevo usuario en la API.

Endpoint: https://api.example.com/users

Ejemplo:

http
```
POST /users HTTP/1.1
Host: api.example.com
Content-Type: application/json

{
  "name": "Alice Johnson",
  "email": "alice@example.com"
}
```

Respuesta Exitosa (201 Created):

json
```
{
  "id": 3,
  "name": "Alice Johnson",
  "email": "alice@example.com"
}
```

Análisis
La solicitud POST al endpoint /users envía los datos del nuevo usuario en formato JSON.
El servidor crea un nuevo usuario con un ID único y devuelve los detalles del usuario creado.

Verbo DELETE:
Se utiliza para eliminar un recurso existente en el servidor.

En este ejemplo, eliminaremos el usuario con ID 2.

Endpoint: https://api.example.com/users/2

Ejemplo:

http
DELETE /users/2 HTTP/1.1
Host: api.example.com

Respuesta Exitosa (204 No Content):

(No hay contenido en la respuesta, ya que el recurso fue eliminado con éxito)

Análisis
La solicitud DELETE al endpoint /users/2 elimina el usuario con ID 2.
El servidor responde con un código 204 No Content, indicando que el recurso se eliminó con éxito.

4. ¿Es MongoDB una base de datos SQL o NoSQL?

MongoDB es una base de datos NoSQL orientada a documentos. No sigue el modelo relacional de las bases de datos SQL tradicionales. En su lugar, utiliza un modelo de almacenamiento de documentos JSON flexible que se adapta bien a aplicaciones modernas y que no requiere un esquema fijo. MongoDB no se basa en tablas ni columnas, ya que los datos se almacenan como documentos y colecciones.

Detalles de las Diferencias entre NoSQL y SQL:

Tipo de Datos:
NoSQL: Permite una variedad de tipos de datos flexibles, como documentos JSON, pares clave-valor, grafos, etc.
SQL: Utiliza tipos de datos predefinidos, como INTEGER, VARCHAR, DATE, etc.

Esquema:
NoSQL: Esquema flexible y dinámico. No requiere una estructura de tabla predefinida.

SQL: Esquema rígido y predefinido. Requiere definir la estructura de las tablas antes de insertar datos.

Escalabilidad:
NoSQL: Escalabilidad horizontal fácilmente, distribuyendo datos en varios servidores.
SQL: Escalabilidad vertical más común, aumentando la capacidad de un servidor individual.

Transacciones:
NoSQL: Algunas implementaciones de NoSQL no admiten transacciones ACID.
SQL: Las bases de datos SQL son conocidas por sus transacciones ACID.

Lenguaje de Consulta:
NoSQL: No hay un lenguaje de consulta estándar. Cada base de datos NoSQL tiene su propia forma de consulta.
SQL: Utiliza SQL (Structured Query Language), un lenguaje estándar para consultar y manipular datos en bases de datos relacionales.

Ejemplos de Bases de Datos:
NoSQL: MongoDB (Documentos), Cassandra (Columnas), Redis (Clave-Valor), CouchDB (Documentos).
SQL: MySQL, PostgreSQL, SQLite, Microsoft SQL Server, Oracle.

Uso:
NoSQL: Mejor para aplicaciones con grandes cantidades de datos y necesidades de escalabilidad.
SQL: Mejor para aplicaciones con estructura de datos definida y necesidades de transacciones seguras.

Ejemplo de Uso:
NoSQL: Aplicaciones web modernas, Big Data, Internet of Things (IoT), donde la flexibilidad y la escalabilidad son importantes.
SQL: Sistemas de gestión empresarial, aplicaciones transaccionales, sistemas donde la integridad de los datos y las transacciones ACID son críticas.

NOTA:
Este esquema comparativo muestra algunas de las diferencias clave entre las bases de datos NoSQL y SQL, cada una con sus ventajas y casos de uso específicos. La elección entre NoSQL y SQL generalmente depende de los requisitos y características específicas de la aplicación y el proyecto.

Fuentes la información:

Documentación Oficial de las Bases de Datos:
MongoDB (NoSQL): Documentación oficial de MongoDB
MySQL (SQL): Documentación oficial de MySQL
PostgreSQL (SQL): Documentación oficial de PostgreSQL

Sitios Web y Blogs de Tecnología:
Medium: Muchos profesionales y expertos en tecnología escriben artículos sobre bases de datos en Medium.
Towards Data Science: Plataforma en Medium con artículos sobre ciencia de datos y bases de datos.
Stack Overflow: Preguntas y respuestas de programadores y desarrolladores sobre bases de datos.

Libros sobre Bases de Datos:
"MongoDB: The Definitive Guide" de Kristina Chodorow para MongoDB.
"Learning MySQL" de Seyed M.M. (Saied) Tahaghoghi y Hugh E. Williams para MySQL.

"PostgreSQL: Up and Running" de Regina Obe y Leo Hsu para PostgreSQL.

Documentos Técnicos y Whitepapers:
Documentos técnicos de las propias empresas de bases de datos: Muchas empresas que desarrollan bases de datos publican documentos técnicos detallados sobre sus productos.
Whitepapers sobre Bases de Datos: Muchas organizaciones, incluidas las empresas de consultoría y análisis de tecnología, publican whitepapers sobre bases de datos.

Sitios Oficiales de Desarrolladores y Comunidades:
GitHub: Repositorios de código y documentación sobre bases de datos.
Foros de Desarrolladores: Como Stack Overflow y Dev.to.

5. ¿Qué es una API?

Una API (Application Programming Interface) es un conjunto de reglas y protocolos que permite a distintos sistemas comunicarse entre sí. En el contexto de desarrollo de software, una API define las funciones y métodos que otros programas pueden utilizar para interactuar con él de manera segura y predecible.

¿Por qué son importantes?
Las APIs (Interfaces de Programación de Aplicaciones) son fundamentales en la tecnología actual por lo cual su demanda sigue creciendo en diversas industrias.

Razones de su importancia:
Interconexión de Aplicaciones y Servicios:
Permiten la integración y comunicación entre aplicaciones y servicios de manera eficiente.
Facilitan la interacción entre sistemas heterogéneos, permitiendo a las aplicaciones compartir y acceder a datos y funcionalidades.

Desarrollo de Ecosistemas y Plataformas:
Ayudan a crear ecosistemas y plataformas más amplias donde múltiples aplicaciones y servicios pueden colaborar y funcionar juntos.
Promueven la creación de ecosistemas de desarrolladores, donde terceros pueden construir sobre una plataforma existente.

Innovación y Desarrollo Ágil:
Aceleran el proceso de desarrollo al permitir a los desarrolladores reutilizar y combinar funcionalidades existentes a través de las APIs.
Facilitan el desarrollo ágil y la implementación de nuevas características y servicios.

Facilitan la Experiencia del Usuario:
Mejoran la experiencia del usuario al permitir la integración de servicios externos.
Posibilitan la creación de aplicaciones y servicios más completos y ricos en funcionalidades.

Acceso a Datos y Funcionalidades:
Permiten a las aplicaciones acceder a datos y funcionalidades de otros sistemas de manera segura y controlada.

Posibilitan la creación de aplicaciones multiplataforma que pueden funcionar en diferentes dispositivos y sistemas operativos.

Ventajas del Uso de APIs:
Reutilización de Funcionalidades:
Fuente: ProgrammableWeb
Las APIs permiten a los desarrolladores reutilizar funciones y servicios existentes, evitando la necesidad de reinventar la rueda. Esto ahorra tiempo y esfuerzo en el desarrollo de nuevas aplicaciones.

Facilitan la Innovación y Expansión:
Fuente: Forbes
Las APIs fomentan la innovación al permitir a las empresas integrar fácilmente nuevas tecnologías y servicios externos en sus propias aplicaciones. Esto impulsa la expansión y la adaptabilidad a las cambiantes demandas del mercado.

Mejora la Experiencia del Cliente:
Fuente: Salesforce
Al permitir la integración de diferentes servicios, las APIs mejoran la experiencia del cliente al ofrecer aplicaciones más completas y personalizadas. Esto conduce a una mayor satisfacción del usuario y fidelización.

Incremento de la Eficiencia y Productividad:
Fuente: Forbes
Las APIs automatizan procesos y tareas al permitir la conexión y transferencia de datos entre sistemas. Esto aumenta la eficiencia operativa y la productividad de las empresas.

Acelera el Tiempo de Comercialización:
Fuente: ProgrammableWeb
Al permitir el acceso rápido a funcionalidades y servicios externos, las APIs aceleran el desarrollo y lanzamiento de nuevas aplicaciones al mercado.

Escalabilidad y Flexibilidad:
Fuente: Forbes
Las APIs ofrecen una arquitectura flexible que permite a las empresas escalar y adaptarse rápidamente a las demandas del mercado y a los cambios en las necesidades de los usuarios.

Fuentes:
ProgrammableWeb - 7 Reasons Why Developers Love APIs
[Forbes - How APIs Drive Business Growth and Digital Transformation](https://www.forbes.com

6. ¿Qué es Postman?
Postman es una herramienta que se utiliza para probar APIs, especialmente APIs RESTful. Permite a los desarrolladores enviar solicitudes HTTP a un servidor y recibir respuestas.

También facilita la creación y organización de solicitudes HTTP, la automatización de pruebas y la documentación de APIs.

Ventajas del uso de Postman en la actualidad

Facilita el Desarrollo de APIs:
Postman simplifica el proceso de desarrollo de APIs al proporcionar una interfaz gráfica intuitiva para crear, probar y depurar endpoints de APIs de manera eficiente.
Fuente: Postman Blog

Pruebas Automatizadas:
Permite escribir y ejecutar pruebas automatizadas para verificar el comportamiento de las APIs, lo que mejora la calidad y confiabilidad del código.
Fuente: Postman Blog

Documentación de APIs:
Facilita la generación de documentación clara y detallada de las APIs, lo que ayuda a los desarrolladores a comprender y utilizar las APIs de manera efectiva.
Fuente: Postman Blog

Colaboración y Compartición de Colecciones:
Postman permite a los equipos colaborar fácilmente en el desarrollo de APIs al compartir colecciones de solicitudes y pruebas.
Fuente: Postman Blog

Ambiente de Pruebas Configurable:
Permite configurar diferentes entornos de pruebas (development, testing, production) para simular diferentes configuraciones y escenarios.
Fuente: Postman Blog

Conclusión
Postman es una herramienta integral que proporciona numerosas ventajas para el desarrollo, prueba y documentación de APIs en la actualidad. Algunas de las ventajas clave incluyen su capacidad para simplificar el desarrollo de APIs, automatizar pruebas, generar documentación clara, facilitar la colaboración entre equipos, y ofrecer un ambiente de pruebas configurable. Estas ventajas se traducen en un proceso de desarrollo más eficiente, mayor calidad del código y una mejor experiencia para los desarrolladores y usuarios de las APIs.

Otras Fuentes:
Postman Blog - Why Use Postman?
DZone - 10 Reasons Why Postman Is A Must For Every Developer
Blazemeter - Why Use Postman?
Apiumhub - 5 Reasons to Use Postman for API Testing
QAInsights - Why You Should Use Postman for API Testing
Estas fuentes proporcionan información detallada y actualizada sobre las ventajas del uso de Postman en el desarrollo y prueba de APIs en la actualidad.

7. ¿Qué es el polimorfismo?

El polimorfismo es un concepto de la programación orientada a objetos, este término "polimorfismo" tiene su origen de las palabras "poly" que significa "muchos" y "morfo" con el significado de "formas", que aplicado a programación hace referencia a los objetos que pueden tener diferentes formas, por lo tanto permite a objetos de diferentes clases responder a métodos de la misma manera. Es decir, un mismo método puede comportarse de manera distinta según el objeto que lo esté ejecutando. Esto permite escribir código más genérico y reutilizable.
Este término tiene relación desde el punto de vista de Python con las palabras en inglés "duck typing", que hacen referencia a la también frase "If it walks like a duck and it quacks like a duck, then it must be a duck" en español es: "Si camina como un pato y habla como un pato, entonces tiene que ser un pato", que en principios base de Python nos dan a entender que "los patos son objetos" y "hablar/andar son métodos", que en palabras de Python se traduce como que si un determinado objeto posee los métodos que nos interesan, eso nos basta, siendo su tipo algo irrelevante.

Ejemplo 1:

```
class Animal:
    def hablar(self):
        pass

class Perro(Animal):
    def hablar(self):
        return "¡Guau!"

class Gato(Animal):
    def hablar(self):
        return "¡Miau!"

# Función genérica que llama al método hablar
def hacer_sonar(animal):
    return animal.hablar()

perro = Perro()
gato = Gato()

print(hacer_sonar(perro))  # Salida: ¡Guau!
print(hacer_sonar(gato))   # Salida: ¡Miau!

# Output
¡Guau!
¡Miau!
```

Ejemplo 2:
Clases para Diferentes Tipos de Empleados

Imaginemos que tenemos un sistema para manejar empleados en una compañía, y queremos calcular sus pagos de manera diferente dependiendo del tipo de empleado: algunos son empleados por hora y otros son empleados con salario fijo.

```
class Empleado:
    def __init__(self, nombre):
        self.nombre = nombre

    def calcular_pago(self):
        pass

    def __str__(self):
        return f"{self.nombre} gana {self.calcular_pago()}"

class EmpleadoPorHora(Empleado):
    def __init__(self, nombre, horas_trabajadas, tarifa_por_hora):
        super().__init__(nombre)
        self.horas_trabajadas = horas_trabajadas
        self.tarifa_por_hora = tarifa_por_hora

    def calcular_pago(self):
        return self.horas_trabajadas * self.tarifa_por_hora

class EmpleadoSalariado(Empleado):
    def __init__(self, nombre, salario_mensual):
        super().__init__(nombre)
        self.salario_mensual = salario_mensual

    def calcular_pago(self):
        return self.salario_mensual

# Lista de empleados de diferentes tipos
empleados = [
    EmpleadoPorHora("Juan", 40, 15),
    EmpleadoSalariado("Ana", 3000),
    EmpleadoPorHora("Luis", 35, 12)
]

for empleado in empleados:
    print(empleado)  # Se imprime el pago de cada empleado, usando polimorfismo

# Output
Juan gana 600
Ana gana 3000
Luis gana 420
```

Análisis del código

Clase Base (Empleado):

Define la interfaz común (calcular_pago), que será implementada de forma diferente por cada subclase.
Subclases (EmpleadoPorHora y EmpleadoSalariado):
Cada una implementa el método calcular_pago de manera que refleje su lógica de pago particular. Esto permite tratar a todos los empleados de manera uniforme a través de su clase base.

Uso de Polimorfismo:
Al llamar a calcular_pago, Python automáticamente ejecuta la implementación correcta del método para cada tipo de empleado. Esto demuestra cómo diferentes objetos pueden ser tratados de forma polimórfica.
Este ejemplo ilustra cómo el polimorfismo facilita la expansión del código y permite que nuevas clases de empleados sean añadidas sin alterar el código que utiliza la clase base.

8. ¿Qué es un método dunder?

Los métodos dunder (double underscore), también conocidos como métodos mágicos, son métodos especiales en Python que comienzan y terminan con doble guion bajo __. Estos métodos tienen un significado especial y son utilizados por Python en circunstancias específicas. Por ejemplo, __init__ para el constructor, __str__ para representar un objeto como cadena, etc.

Ejemplo 1:

```
class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

    def __str__(self):
        return f"{self.titulo} por {self.autor}"

libro = Libro("Python for Beginners", "John Doe")
print(libro)  # Salida: Python for Beginners por John Doe

# Output
Python for Beginners por John Doe
```

Ejemplo 2:

Supongamos que queremos definir una clase que representa un vector en un plano bidimensional y queremos poder sumar dos vectores usando el operador +

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def __add__(self, other):
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        return NotImplemented

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

# Crear dos vectores
v1 = Vector(2, 4)
v2 = Vector(5, -1)

# Sumar dos vectores usando el operador +
v3 = v1 + v2

print(v3)

# Output:
Vector(7, 3)
```

Análisis del código
__init__:
Constructor de la clase que inicializa los componentes "x" y también "y" del vector.

__add__:
Método dunder que se invoca cuando usas el operador + con instancias de Vector. En este método, sumamos los componentes correspondientes de dos vectores y retornamos un nuevo vector resultante.

__repr__:
Método dunder utilizado para obtener la representación de cadena de la instancia de la clase, útil para depuración y registro.

Este ejemplo ilustra cómo podemos sobrecargar el operador + para que funcione con nuestras clases personalizadas, proporcionando una forma intuitiva y matemáticamente correcta de sumar vectores. Esto hace que el código sea más limpio y legible, y es un buen ejemplo de cómo los métodos dunder permiten que las instancias de clases personalizadas se comporten como objetos integrados en Python.

9. ¿Qué es un decorador de Python?

Un decorador en Python es una función que toma otra función como argumento y extiende su funcionalidad sin modificarla explícitamente. Los decoradores son útiles para agregar funcionalidades a funciones existentes de manera transparente.

Ejemplo 1:

```
def decorador(funcion):
    def wrapper():
        print("Antes de llamar a la función.")
        funcion()
        print("Después de llamar a la función.")
    return wrapper

@decorador
def saludo():
    print("Hola, mundo!")

saludo()


# Output
Antes de llamar a la función.
Hola, mundo!
Después de llamar a la función.
```

Análisis del código:
En este ejemplo, decorador es un decorador que agrega mensajes antes y después de llamar a la función saludo. Al usar @decorador encima de la definición de saludo, aplicamos ese decorador a la función saludo.

Ejemplo 2:

Decorador en Python: Medición de Tiempo de Ejecución
Este decorador es muy práctico para entender cómo funciona la decoración y para obtener información valiosa sobre la ejecución de tus programas.

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()  # Guarda el tiempo de inicio
        result = func(*args, **kwargs)  # Ejecuta la función
        end_time = time.time()  # Tiempo después de ejecutar la función
        print(f"{func.__name__} se ejecutó en: {end_time - start_time} segundos")
        return result
    return wrapper

@timer
def suma_numeros(n):
    total = 0
    for num in range(1, n + 1):
        total += num
    return total
```

```
# Usar la función decorada
resultado = suma_numeros(10000)
print(f"Suma total: {resultado}")

# Output
suma_numeros se ejecutó en: 0.0 segundos
Suma total: 50005000
```

Análisis del código:
timer: Es el decorador que toma una función func como argumento.
wrapper: Es una función interna que realiza la medición del tiempo. Guarda el tiempo antes de ejecutar la función, llama a la función con los argumentos que recibe, y luego mide el tiempo nuevamente después de que la función ha terminado.
Decorando suma_numeros: Aplicamos el decorador @timer a la función suma_numeros, que simplemente suma todos los números hasta n.

¿Cómo funciona?
Cuando ejecutas suma_numeros(10000), no llamas directamente a suma_numeros, sino a wrapper, que es la función que se ha envuelto alrededor de suma_numeros gracias al decorador. wrapper mide el tiempo, ejecuta suma_numeros, y luego imprime cuánto tiempo tomó ejecutar la función.

Este ejemplo es ideal para principiantes porque combina conceptos fundamentales (bucles y sumas) con el concepto más avanzado de decoradores, mostrando de manera práctica cómo Python permite extender y modificar el comportamiento de funciones sin modificar directamente su código fuente.