CHECKPOINT 6

¿Para qué usamos Clases en Python?

Las clases en Python son una herramienta fundamental en la programación orientada a objetos (OOP, por sus siglas en inglés). Proporcionan una forma de organizar y estructurar tu código de una manera más modular y reutilizable.

¿Qué son las clases en Python?

Una clase es una plantilla para crear objetos. Contiene atributos (variables) y métodos (funciones) que definen el comportamiento y las características de esos objetos. Los objetos son instancias de una clase, lo que significa que son ejemplos específicos que se crean a partir de esa plantilla.

¿Por qué usar clases en Python?

1. **Abstracción**: Las clases permiten abstraer los datos y las operaciones relacionadas con esos datos en un solo lugar, lo que facilita el manejo y la comprensión del código.
2. **Reutilización de código**: Una vez que has definido una clase, puedes crear múltiples objetos basados en esa clase en diferentes partes de tu programa. Esto fomenta la reutilización del código y ayuda a evitar la duplicación.
3. **Modularidad**: Las clases permiten dividir un programa en módulos más pequeños y manejables. Cada clase puede representar un componente o una entidad del problema que estás resolviendo.
4. **Encapsulación**: Las clases pueden ocultar los detalles internos de implementación y exponer solo una interfaz pública. Esto ayuda a prevenir modificaciones accidentales y a facilitar el mantenimiento del código.

Ejemplo de clase en Python:

Supongamos que queremos modelar un "Perro" en nuestro programa.

```
class Perro:

    def __init__(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad
```

```
    def ladrar(self):
        return f"{self.nombre} está ladrando!"
```

En este ejemplo:

- `Perro` es el nombre de la clase.
- `__init__` es un método especial llamado constructor que se llama automáticamente cuando se crea un nuevo objeto de la clase. Permite inicializar los atributos de la clase.
- `self` es una referencia al objeto actual. Se usa para acceder a los atributos y métodos del objeto dentro de la clase.
- `nombre` y `edad` son atributos de la clase.
- `ladrar` es un método de la clase que devuelve un mensaje indicando que el perro está ladrando.

Buenas prácticas al trabajar con clases en Python:

1. **Seguir convenciones de nomenclatura**: Nombra las clases usando CamelCase (primeras letras en mayúsculas) y utiliza nombres descriptivos para los métodos y atributos.
2. **Usar el principio de responsabilidad única**: Cada clase debe tener una única responsabilidad o tarea.
3. **Documentar tu código**: Usa docstrings para documentar tus clases, métodos y atributos.
4. **Seguir el principio DRY (Don't Repeat Yourself)**: Evita la duplicación de código definiendo clases y reutilizando métodos y atributos.
5. **Mantén el código limpio y legible**: Usa la indentación adecuada y sigue las convenciones de estilo de Python (PEP 8).

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

El método que se ejecuta automáticamente cuando se crea una instancia de una clase en Python es el método especial `__init__()`, también conocido como el constructor de la clase. Este método se define dentro de la clase y se llama automáticamente cuando se crea un nuevo objeto de esa clase. Su propósito principal es inicializar los atributos del objeto.

El siguiente es un ejemplo para ilustrar cómo se utiliza el método `__init__()`:

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

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

# El método __init__() se ejecutó automáticamente al crear persona1
```

En este ejemplo, cuando se crea la instancia `persona1` de la clase `Persona`, se llama automáticamente al método `__init__()`, pasando los argumentos `"Juan"` y `30` para inicializar los atributos `nombre` y `edad` del objeto `persona1`.

¿Cuáles son los tres verbos de API?

Los tres verbos principales utilizados en la arquitectura RESTful para interactuar con recursos a través de una API son:

1. **GET**: Este verbo se utiliza para solicitar datos de un recurso específico o de una colección de recursos. Por lo general, se utiliza para recuperar información y no debe tener un efecto secundario en el servidor.
2. **POST**: Se utiliza para enviar datos al servidor para crear un nuevo recurso. Por lo general, se emplea cuando se quiere agregar un nuevo elemento a una colección de recursos.
3. **DELETE**: Este verbo se utiliza para eliminar un recurso específico en el servidor. Se envía una solicitud con el método DELETE al recurso que se desea eliminar.

Estos tres verbos forman la base de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en una API RESTful, permitiendo la manipulación de recursos a través de solicitudes HTTP. Además de estos, también existen otros verbos HTTP como PUT (para actualizar recursos), PATCH (para modificar parcialmente un recurso) y OPTIONS (para obtener información sobre los

métodos permitidos en un recurso), pero los tres mencionados anteriormente son los más comunes en la mayoría de las API RESTful.

¿Es MongoDB una base de datos SQL o NoSQL?

MongoDB es una base de datos NoSQL. NoSQL se refiere a una amplia categoría de sistemas de gestión de bases de datos que no se basan en el modelo relacional de las bases de datos SQL tradicionales. En cambio, NoSQL (que significa "Not Only SQL") se caracteriza por su flexibilidad en esquemas, escalabilidad horizontal y capacidades de procesamiento de grandes volúmenes de datos no estructurados.

MongoDB es una base de datos NoSQL de tipo documento, lo que significa que almacena datos en forma de documentos similares a JSON (JavaScript Object Notation). Los documentos son estructuras de datos flexibles que pueden contener pares de clave-valor y arrays. Este modelo de datos flexible hace que MongoDB sea adecuado para una amplia variedad de casos de uso, incluidos aquellos en los que los esquemas de datos cambian con frecuencia o en los que se requiere una rápida iteración en el desarrollo.

En resumen, MongoDB es una base de datos NoSQL que se ha vuelto popular debido a su escalabilidad, flexibilidad y capacidad para manejar datos semi-estructurados y no estructurados.

¿Qué es una API?

Una API (Application Programming Interface, por sus siglas en inglés) es un conjunto de reglas y definiciones que permite que diferentes aplicaciones se comuniquen entre sí. En términos simples, una API define cómo interactuar con un sistema de software específico, proporcionando un conjunto de funciones, métodos y protocolos que permiten a otras aplicaciones utilizar las funcionalidades de ese sistema de manera controlada y segura.

Las APIs son esenciales en el desarrollo de software porque permiten la integración de diferentes sistemas, la reutilización de código y la construcción de aplicaciones modulares y escalables. Algunos ejemplos comunes de APIs incluyen:

1. **APIs web**: Permiten la comunicación entre sistemas a través de Internet utilizando protocolos como HTTP o HTTPS. Las APIs web son utilizadas

para acceder a servicios en línea, como redes sociales, servicios de pago, servicios de mapas, entre otros.
2. **APIs de bibliotecas de programación**: Son conjuntos de funciones y métodos proporcionados por bibliotecas de software para permitir a los desarrolladores utilizar ciertas funcionalidades en sus aplicaciones. Por ejemplo, la API de la biblioteca estándar de Python proporciona funciones para manejar archivos, realizar operaciones matemáticas, interactuar con el sistema operativo, entre otras tareas.
3. **APIs de sistemas operativos**: Permiten a las aplicaciones interactuar con el sistema operativo subyacente para acceder a recursos del sistema, como el sistema de archivos, el hardware, la red, etc.

En resumen, una API actúa como un intermediario que define cómo se pueden utilizar las funcionalidades de un sistema de software específico, facilitando la comunicación y la integración entre diferentes aplicaciones y componentes de software.

¿Qué es Postman?

Postman es una plataforma colaborativa para el desarrollo de API que facilita la creación, prueba, documentación y compartición de APIs. Originalmente, Postman era una herramienta para realizar solicitudes HTTP y probar API, pero ha evolucionado para convertirse en una suite completa de herramientas para todo el ciclo de vida de desarrollo de API.

Algunas características principales de Postman incluyen:

1. **Interfaz de usuario intuitiva**: Postman proporciona una interfaz de usuario fácil de usar que permite a los desarrolladores crear y enviar solicitudes HTTP a APIs, ver respuestas, inspeccionar encabezados y datos de respuesta, y más.
2. **Colecciones**: Los desarrolladores pueden organizar sus solicitudes en colecciones, lo que facilita la gestión y la ejecución de múltiples solicitudes relacionadas.
3. **Entorno**: Postman permite definir entornos, que son conjuntos de variables que se pueden utilizar para configurar y personalizar las solicitudes. Esto es útil para probar APIs en diferentes entornos, como desarrollo, pruebas o producción.
4. **Pruebas automáticas**: Postman permite escribir scripts de pruebas automáticas usando JavaScript para validar las respuestas de las APIs. Esto es útil para automatizar la prueba y verificación del comportamiento de las APIs.

5. **Documentación de API**: Postman permite generar documentación interactiva para APIs directamente desde las colecciones de solicitudes, lo que facilita la comprensión y el uso de las APIs por parte de otros desarrolladores.
6. **Colaboración**: Postman ofrece características de colaboración que permiten a los equipos trabajar juntos en el desarrollo, la prueba y la documentación de APIs.

En resumen, Postman es una herramienta poderosa y versátil que ayuda a los desarrolladores a trabajar de manera más eficiente en el desarrollo de APIs, desde la creación y la prueba hasta la documentación y la colaboración.

¿Qué es el polimorfismo?

El polimorfismo es un concepto clave en la programación orientada a objetos (OOP) que se refiere a la capacidad de objetos de diferentes clases de responder de manera diferente a la misma invocación de método. En otras palabras, el polimorfismo permite tratar objetos de diferentes clases de manera uniforme siempre que implementen cierto comportamiento común.

Hay dos tipos principales de polimorfismo en la OOP:

1. **Polimorfismo de subtipos (o herencia)**: Este tipo de polimorfismo se basa en la relación de herencia entre clases. Se refiere a la capacidad de una clase base (o superclase) de ser referenciada por una referencia de su subclase. Cuando se llama a un método en un objeto de la superclase, el comportamiento que se ejecuta puede ser el de la subclase si la subclase ha sobreescrito ese método. Esto permite que diferentes tipos de objetos respondan de manera diferente a la misma invocación de método.
2. **Polimorfismo de inclusión (o interfaces)**: Este tipo de polimorfismo se basa en la implementación de interfaces o protocolos. Las interfaces definen un conjunto de métodos que una clase debe implementar. Diferentes clases pueden implementar la misma interfaz, lo que permite tratar objetos de diferentes clases de manera uniforme si implementan la misma interfaz. En este caso, el polimorfismo se logra a través de la inclusión de múltiples clases en un conjunto común de operaciones definidas por una interfaz.

El polimorfismo es una característica importante de la programación orientada a objetos porque permite escribir código más genérico y flexible, lo que facilita la reutilización y la extensibilidad del código. Además, promueve la

interoperabilidad entre diferentes partes de un sistema de software al permitir que objetos de diferentes clases interactúen de manera coherente.

¿Qué es un método dunder?

Un "método dunder" es un término informal que se refiere a los métodos especiales en Python que tienen nombres que comienzan y terminan con doble guion bajo (también conocidos como "métodos mágicos" o "métodos especiales"). Estos métodos son proporcionados por Python para permitir la personalización y la definición de comportamientos específicos para objetos en diferentes situaciones.

Los métodos dunder son invocados automáticamente por Python en ciertas operaciones, como la creación de objetos, la realización de operaciones aritméticas, la comparación de objetos, etc. Al implementar estos métodos en una clase, puedes modificar el comportamiento predeterminado de los operadores y las funciones incorporadas de Python para tus propios tipos de datos personalizados.

Algunos ejemplos de métodos dunder comunes incluyen:

- `__init__`: Método constructor, llamado automáticamente al crear una nueva instancia de la clase.
- `__str__`: Método que devuelve una representación legible como cadena del objeto, llamado automáticamente cuando se utiliza la función `str()`.
- `__add__`, `__sub__`, `__mul__`, etc.: Métodos para realizar operaciones aritméticas, llamados automáticamente al utilizar los operadores `+`, `-`, `*`, etc.
- `__eq__`, `__ne__`, `__lt__`, `__gt__`, etc.: Métodos para comparar objetos, llamados automáticamente al utilizar los operadores de comparación como `==`, `!=`, `<`, `>`, etc.

Los métodos dunder son una parte integral de la programación orientada a objetos en Python y son utilizados para hacer que tus clases sean más poderosas y flexibles al personalizar su comportamiento en diferentes contextos.

¿Qué es un decorador de python?

Un decorador en Python es una función especial que se utiliza para modificar o extender el comportamiento de otras funciones o métodos. En esencia, un decorador toma una función existente y la envuelve en otra función, permitiendo agregar funcionalidades adicionales sin modificar directamente el código de la función original.

Los decoradores se utilizan comúnmente para aspectos como la validación de datos, el registro de funciones, la gestión de excepciones, la autenticación, el control de acceso, la medición del tiempo de ejecución y más. Ayudan a mantener el código limpio, modular y reutilizable al permitir la separación de preocupaciones.

En Python, los decoradores se implementan utilizando una sintaxis especial con el símbolo `@` seguido del nombre del decorador encima de la definición de la función que se desea decorar. Aquí un ejemplo básico de cómo se ve la sintaxis de un decorador:

```
def mi_decorador(funcion):

    def wrapper(*args, **kwargs):

        # Aquí puedes agregar funcionalidades adicionales antes y/o después de llamar a la función original

        print("Ejecutando el decorador antes de la función")

        resultado = funcion(*args, **kwargs)

        print("Ejecutando el decorador después de la función")

        return resultado

    return wrapper



@mi_decorador

def mi_funcion():

    print("¡Hola, mundo!")
```

```
mi_funcion()
```

En este ejemplo, **`mi_decorador`** es el decorador que envuelve a **`mi_funcion`**. Cuando se llama a **`mi_funcion`**, en realidad se está llamando a **`wrapper`**, que ejecuta el código adicional definido en el decorador antes y después de llamar a **`mi_funcion`**.

Los decoradores son una poderosa herramienta en Python que permite añadir funcionalidades adicionales de manera elegante y modular. Son ampliamente utilizados en muchos frameworks y bibliotecas de Python para extender y personalizar su funcionalidad.