Full Stack Development
Checkpoint 6
Mohamed ALwassy
Fundacion Vass
2024

Porque usamos Clases en Python?

En Python, las clases se utilizan principalmente para definir nuevos tipos de objetos(un objeto es una copia de la clase que hereda todo el codigo de la clase), proporcionando una forma de encapsular datos y funcionalidad relacionada. Aquí hay algunas razones específicas por las que se utilizan clases en Python:

1. **Encapsulación**: Las clases permiten agrupar datos (atributos) y métodos (funciones) que operan sobre esos datos en una sola unidad. Esto ayuda a organizar el código y a mantenerlo modular.
2. **Reutilización de código**: Las clases permiten la creación de plantillas para objetos. Una vez definida una clase, se pueden crear múltiples instancias de esa clase, lo que facilita la reutilización del código.
3. **Herencia**: Las clases permiten heredar propiedades y métodos de otras clases. Esto facilita la creación de nuevas clases que reutilizan y extienden la funcionalidad de clases existentes, promoviendo así la reutilización y extensión del código.
4. **Polimorfismo**: Permite que las funciones usen objetos de diferentes clases de manera uniforme. Esto se logra definiendo métodos en la clase base que pueden ser sobreescritos en las clases derivadas.
5. **Abstracción**: Las clases permiten ocultar detalles complejos y exponer solo lo necesario a través de interfaces claras. Esto simplifica el uso de objetos complejos.

   Ejemplo:

En este ejemplo hemos creado una clase llamada saludar con el método class de python, en el cual hemos definido dos funciones la primera es el método INIT que es un constructor y su función es asignar valores iniciales a los atributos del objeto cuando se crea una nueva estancia, y que en nuestro caso lleva tres atributos self que se usa para acceder internamente a los atributos

y métodos de la misma clase y nombre y edad. Abajo tenemos la función saludo la cual lleva el atributo self tambien y que devuelve un mensaje del saludo cuando es llamada.

Debajo hemos creado dos instancias nuevas persona1 y persona 2 de la misma clase saludar las cuales hemos asignado a cada una parametros diferentes, y como las dos instancias heredan de la clase saludar podemos acceder y utilizar la función saludo y obtener resultados diferentes como puedes observar a la derecha del codigo.

¿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 es el método __init__ que es el constructor y su función es inicializar los atributos del objeto recién creado, y no devuelve ningún valor.
En la definición de una clase suele haber un método llamado `__init__` que se conoce como *inicializador*. Este método es un método especial que se llama cada vez que se instancia una clase y sirve para inicializar el objeto que se crea. Este método crea los atributos que deben tener todos los objetos de la clase y por tanto contiene los parámetros necesarios para su creación, pero no devuelve nada. Se invoca cada vez que se instancia un objeto de esa clase.
Ejemplo:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre  # Inicializa el atributo nombre
        self.edad = edad      # Inicializa el atributo edad
```

Como se puede ver en este ejemplo hemos creado una clase llamada Persona y en ella hemos creado el método __init__ con tres parametros: self para acceder internamente a los parámetros de la clase que es igual al método this en Java, y hemos inicializado los atributos nombre y edad.

¿Cuáles son los tres verbos de API?

Los tres verbos principales de una API (Interfaz de Programación de Aplicaciones),son:

1. **GET**: Este verbo se utiliza para **recuperar datos** de un servidor. No modifica ningún dato en el servidor, solo solicita información y devuelve la respuesta. Es idempotente, lo que significa que realizar la misma solicitud varias veces tendrá el mismo efecto que hacerlo una sola vez.
   Ejemplo:
   - `GET /usuarios` podría devolver una lista de todos los usuarios.
   - `GET /usuarios/1` podría devolver los detalles del usuario con ID 1.

2. **POST**: Este verbo se utiliza para **crear nuevos recursos** en el servidor. Envía datos al servidor para que se creen nuevos registros. No es idempotente, lo que significa que realizar la misma solicitud varias veces puede resultar en la creación de múltiples recursos.
   Ejemplo:
   - `POST /usuarios` con un cuerpo de solicitud que contenga los datos del usuario (como nombre, correo electrónico, etc.) creará un nuevo usuario.

3. **PUT**: Este verbo se utiliza para **actualizar completamente un recurso existente** en el servidor. Envía datos al servidor para actualizar un recurso específico. Es idempotente, lo que significa que realizar la misma solicitud varias veces tendrá el mismo efecto que hacerlo una sola vez, actualizando el recurso al estado definido por la solicitud.
   Ejemplo:
   - `PUT /usuarios/1` con un cuerpo de solicitud que contenga los datos del usuario (como nombre, correo electrónico, etc.) actualizará el usuario con ID 1.

Además de estos tres verbos principales, hay otros verbos importantes en el contexto de las API REST:

- **DELETE**: Se utiliza para **eliminar un recurso** existente en el servidor. Es idempotente, ya que eliminar un recurso inexistente no tiene efecto adicional.
  Ejemplo:
  - `DELETE /usuarios/1` eliminará el usuario con ID 1.

- **PATCH**: Se utiliza para **actualizar parcialmente un recurso** existente. Solo cambia los datos especificados en la solicitud. No es necesariamente idempotente dependiendo de cómo se implemente.
  Ejemplo:
  - `PATCH /usuarios/1` con un cuerpo de solicitud que contenga solo el campo de correo electrónico actualizará solo el correo electrónico del usuario con ID 1.

Estos verbos permiten a las API proporcionar una interfaz completa para interactuar con los datos del servidor de manera coherente y estandarizada.

¿Es MongoDB una base de datos SQL o NoSQL?

MongoDB es una base de datos NoSQL. A diferencia de las bases de datos SQL que utilizan un modelo relacional basado en tablas y esquemas predefinidos,

MongoDB emplea un modelo de datos basado en documentos. En MongoDB, los datos se almacenan en documentos similares a JSON (JavaScript Object Notation), que son más flexibles y permiten almacenar datos anidados y estructurados de manera más dinámica.

Aquí hay algunas características clave de MongoDB como una base de datos NoSQL:

1. **Modelo de Documentos**: Utiliza documentos BSON (una versión binaria de JSON) para almacenar datos, lo que permite una representación rica y jerárquica de la información.
2. **Esquema Flexible**: A diferencia de las bases de datos SQL, no requiere esquemas predefinidos, lo que facilita la modificación y adaptación de la estructura de los datos a medida que evolucionan las necesidades de la aplicación.
3. **Escalabilidad Horizontal**: Diseñada para escalar fácilmente en múltiples servidores, lo que la hace adecuada para aplicaciones con grandes volúmenes de datos y necesidades de alto rendimiento.
4. **Consultas Avanzadas**: Ofrece potentes capacidades de consulta, indexación y agregación que permiten operaciones complejas sobre los datos almacenados.

Estas características hacen de MongoDB una opción popular para aplicaciones que requieren flexibilidad en el modelado de datos, capacidad de manejar grandes volúmenes de datos y la necesidad de escalabilidad.

Aquí tienes un ejemplo sencillo para ilustrar cómo se usa MongoDB en comparación con una base de datos SQL tradicional.

Ejemplo en SQL

Supongamos que tenemos una base de datos para gestionar información sobre libros y autores. En SQL, podríamos tener dos tablas: `authors` y `books`.

Tabla `authors`:

id	name
1	J.K. Rowling
2	J.R.R. Tolkien

Tabla `books`:

id	title	author_id
1	Harry Potter	1
2	The Hobbit	2

Para consultar los libros y sus autores, utilizaríamos una sentencia SQL como:

```sql
SELECT books.title, authors.name
FROM books
JOIN authors ON books.author_id = authors.id;
```

Ejemplo en MongoDB

En MongoDB, la estructura es más flexible y podemos almacenar los datos en un solo documento si lo deseamos. Aquí podríamos tener una colección llamada `books` que almacena documentos como estos:

```json
{
  "_id": 1,
  "title": "Harry Potter",
  "author": {
    "name": "J.K. Rowling"
  }
}
```

json

```
{
  "_id": 2,
  "title": "The Hobbit",
  "author": {
    "name": "J.R.R. Tolkien"
  }
}
```

Para insertar estos documentos en MongoDB, usaríamos comandos como:

```javascript
db.books.insertMany([
  {
    "_id": 1,
    "title": "Harry Potter",
    "author": {
      "name": "J.K. Rowling"
    }
  },
  {
    "_id": 2,
    "title": "The Hobbit",
    "author": {
      "name": "J.R.R. Tolkien"
    }
  }
]);
```

Y para consultar los libros y sus autores, podemos hacer una consulta sencilla:

```javascript
db.books.find({}, { title: 1, "author.name": 1, _id: 0 });
```

Esta consulta devolverá resultados como:

```
[
  {
    "title": "Harry Potter",
    "author": {
      "name": "J.K. Rowling"
    }
  },
```

Comparación

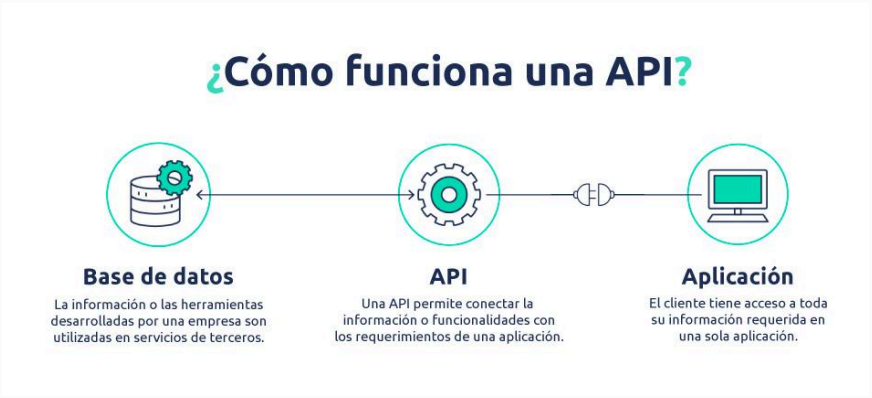
- **Flexibilidad**: En MongoDB, podemos anidar los datos dentro de un documento, eliminando la necesidad de unir tablas como en SQL.
- **Escalabilidad**: MongoDB está diseñado para escalar horizontalmente, lo que es ideal para aplicaciones con grandes volúmenes de datos.
- **Esquema**: MongoDB permite esquemas flexibles, lo que facilita el cambio y evolución de la estructura de datos sin necesidad de alterar un esquema fijo como en SQL.

Esta flexibilidad y escalabilidad hacen que MongoDB sea una opción popular para aplicaciones modernas que requieren agilidad en el manejo de datos.

¿Qué es una API?

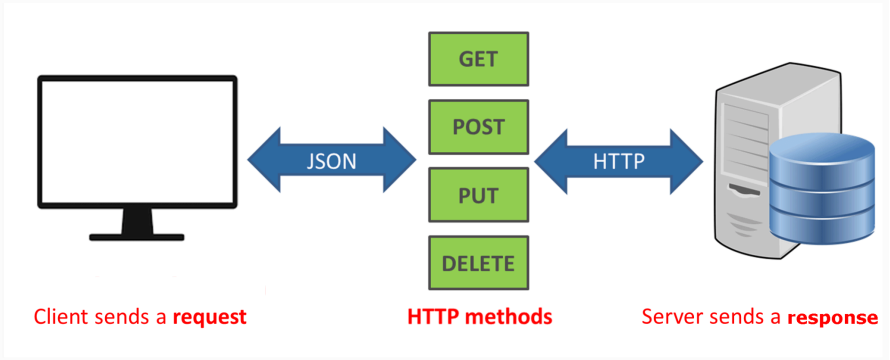
Las API son mecanismos que permiten a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos. Por ejemplo, el sistema de software del instituto de meteorología contiene datos meteorológicos diarios. La aplicación meteorológica de su teléfono “habla” con este sistema a través de las API y le muestra las actualizaciones meteorológicas diarias en su teléfono.

API significa “interfaz de programación de aplicaciones”. En el contexto de las API, la palabra aplicación se refiere a cualquier software con una función distinta. La interfaz puede considerarse como un contrato de servicio entre dos aplicaciones. Este contrato define cómo se comunican entre sí mediante solicitudes y respuestas. La documentación de su API contiene información sobre cómo los desarrolladores deben estructurar esas solicitudes y respuestas.



¿Cómo funcionan las API?

La arquitectura de las API suele explicarse en términos de cliente y servidor. La aplicación que envía la solicitud se llama cliente, y la que envía la respuesta se llama servidor. En el ejemplo del tiempo, la base de datos meteorológicos del instituto es el servidor y la aplicación móvil es el cliente.



¿Qué es Postman?

Postman es una plataforma de colaboración para el desarrollo, prueba y documentación de APIs. Es una herramienta muy popular entre los desarrolladores y equipos de desarrollo por su facilidad de uso y sus numerosas funcionalidades que ayudan a gestionar el ciclo de vida completo de las APIs. Aquí hay una descripción detallada de Postman y sus características:

Características Principales de Postman:

1. **Interfaz Gráfica de Usuario (GUI)**:
   - **Facilidad de Uso**: Postman proporciona una interfaz de usuario intuitiva que permite a los desarrolladores construir, probar y documentar solicitudes HTTP sin necesidad de escribir código.
   - **Colecciones**: Los usuarios pueden organizar sus solicitudes en colecciones, lo que facilita la gestión de múltiples solicitudes y su ejecución secuencial o paralela.
2. **Construcción y Prueba de Solicitudes**:
   - **Tipos de Solicitudes**: Postman soporta varios tipos de solicitudes HTTP, como GET, POST, PUT, DELETE, PATCH, entre otras.
   - **Parámetros y Encabezados**: Los usuarios pueden añadir fácilmente parámetros de consulta, encabezados, cuerpos de solicitudes (en varios formatos como JSON, XML, Form Data), y manejar cookies.
3. **Automatización de Pruebas**:

    - **Scripts de Pre-solicitud y Pruebas**: Postman permite añadir scripts en JavaScript que se ejecutan antes y después de las solicitudes para automatizar pruebas y validaciones.
    - **Monitores**: Los usuarios pueden configurar monitores para ejecutar colecciones de solicitudes en intervalos programados y recibir notificaciones en caso de fallos.
4. **Colaboración y Documentación**:
    - **Colaboración en Equipo**: Postman facilita la colaboración entre equipos permitiendo compartir colecciones, entornos y archivos de configuración.
    - **Documentación**: Las APIs pueden ser documentadas directamente en Postman, y esta documentación puede ser publicada y compartida fácilmente.
5. **Generación de Código**:
    - **Generador de Código**: Postman puede generar código en múltiples lenguajes de programación (como Python, JavaScript, Java, PHP, etc.) para facilitar la integración de las solicitudes en aplicaciones.
6. **Integraciones y Extensiones**:
    - **Integraciones**: Postman se integra con muchas herramientas de desarrollo y CI/CD (como Jenkins, GitHub, GitLab, y más).
    - **Postman API**: Postman ofrece su propia API que permite interactuar programáticamente con sus funcionalidades.

Usos Comunes de Postman

- **Desarrollo de APIs**: Construir y probar las APIs durante el proceso de desarrollo.
- **Depuración**: Identificar y solucionar problemas en las APIs.
- **Automatización de Pruebas**: Realizar pruebas automatizadas y validaciones de las APIs.
- **Documentación**: Crear y mantener documentación actualizada y accesible para las APIs.
- **Monitoreo**: Supervisar el rendimiento y la disponibilidad de las APIs en producción.

Ejemplo Práctico

Supongamos que estás desarrollando una API para una tienda en línea. Puedes usar Postman para:

1. **Crear Solicitudes**: Configurar solicitudes para probar los endpoints de productos, carritos de compras y órdenes.
2. **Automatizar Pruebas**: Escribir scripts de prueba para asegurarte de que los endpoints funcionan como se espera.
3. **Documentar**: Documentar los endpoints y sus funcionalidades para que otros desarrolladores puedan entender y usar tu API.
4. **Monitorear**: Configurar monitores que ejecuten tus pruebas periódicamente para asegurarte de que la API sigue funcionando correctamente en producción.

En resumen, Postman es una herramienta versátil y poderosa que facilita todo el ciclo de vida de las APIs, desde el desarrollo y prueba hasta la documentación y monitoreo, promoviendo la colaboración y eficiencia en los equipos de desarrollo.

¿Qué es el polimorfismo?

El polimorfismo es un concepto fundamental en la programación orientada a objetos (OOP) que permite a objetos de diferentes clases ser tratados como objetos de una clase común. Más específicamente, el polimorfismo se refiere a la capacidad de diferentes objetos de responder a la misma interfaz o método de diferentes maneras. Este concepto es crucial para crear software flexible y reutilizable.

Existen dos tipos principales de polimorfismo:

1. **Polimorfismo en tiempo de compilación (o estático)**: Se logra a través de la sobrecarga de métodos. Esto significa que en una clase, puedes tener múltiples métodos con el mismo nombre pero diferentes firmas (diferentes tipos o números de parámetros). El compilador decide qué método llamar basado en la firma del método durante la compilación.

```java
class Calculadora {
    int sumar(int a, int b) {
        return a + b;
    }

    double sumar(double a, double b) {
        return a + b;
    }
}
```

2. Polimorfismo en tiempo de ejecución (o dinámico): Se logra a través de la herencia y el uso de métodos sobreescritos. En este caso, una referencia de una clase base puede referirse a un objeto de cualquier clase derivada y el método correspondiente que se ejecutará se determina en tiempo de ejecución.

```
class Animal {
    void hacerSonido() {
        System.out.println("Sonido de animal");
    }
}

class Perro extends Animal {
    void hacerSonido() {
        System.out.println("Ladrar");
    }
}

class Gato extends Animal {
    void hacerSonido() {
        System.out.println("Maullar");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Animal miAnimal;
        miAnimal = new Perro();
        miAnimal.hacerSonido(); // Imprime "Ladrar"

        miAnimal = new Gato();
        miAnimal.hacerSonido(); // Imprime "Maullar"
    }
}
```

En resumen, el polimorfismo permite diseñar sistemas más flexibles y extensibles, ya que los objetos pueden ser intercambiados y manipulados de manera uniforme, facilitando la ampliación y modificación del código sin afectar significativamente otras partes del sistema.

¿Qué es un método dunder?

Un método dunder, abreviatura de "double underscore" (doble guion bajo), es un método especial en Python que tiene guiones bajos dobles al principio y al final de su nombre. Estos métodos son también conocidos como métodos mágicos o métodos especiales. Se utilizan para permitir que las clases definan comportamientos específicos en ciertas operaciones integradas del lenguaje, como la representación de cadenas, la adición, la multiplicación, y la indexación, entre otros.

Algunos ejemplos comunes de métodos dunder incluyen:

- `__init__(self, ...)`: Inicializa una nueva instancia de una clase.
- `__str__(self)`: Define el comportamiento para la conversión a cadena de texto cuando se usa `str()`.
- `__repr__(self)`: Define una representación oficial de la cadena de texto de la instancia.
- `__add__(self, other)`: Define el comportamiento para la suma con el operador `+`.

- `__len__(self)`: Define el comportamiento para la función `len()`.
- `__getitem__(self, key)`: Define el comportamiento para la indexación con `[]`.

Estos métodos permiten personalizar el comportamiento de las instancias de las clases para que se comporten de manera específica cuando se usan en distintas operaciones o contextos.

Ejemplo:

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

    def __str__(self):
        return f"MiClase con valor {self.valor}"

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

obj1 = MiClase(10)
obj2 = MiClase(20)

print(obj1)          # Llama a __str__, salida: "MiClase con valor 10"
print(obj1 + obj2)   # Llama a __add__, salida: "MiClase con valor 30"
```

En este ejemplo, `__str__` y `__add__` son métodos dunder que permiten la conversión a cadena y la adición personalizada de instancias de `MiClase`, respectivamente.

¿Qué es un decorador de python?

Un decorador en Python es una función que modifica el comportamiento de otra función o método. Los decoradores permiten añadir funcionalidad a una función o método existente de una manera clara y legible sin modificar directamente su código. Se usan comúnmente para extender el comportamiento de funciones o

métodos en diversos contextos, como la validación de entradas, la gestión de recursos, la autenticación, la medición del tiempo de ejecución, entre otros.

La sintaxis básica de un decorador utiliza el símbolo `@` seguido del nombre del decorador colocado justo antes de la definición de la función que se va a decorar.

Ejemplo:

```python
def mi_decorador(func):
    def nueva_funcion(*args, **kwargs):
        print("Algo que ocurre antes de ejecutar la función")
        resultado = func(*args, **kwargs)
        print("Algo que ocurre después de ejecutar la función")
        return resultado
    return nueva_funcion

@mi_decorador
def saludar():
    print("Hola!")

saludar()
```

Definición del Decorador:

```python
def mi_decorador(func):
    def nueva_funcion(*args, **kwargs):
        print("Algo que ocurre antes de ejecutar la función")
        resultado = func(*args, **kwargs)
        print("Algo que ocurre después de ejecutar la función")
        return resultado
    return nueva_funcion
```

Aquí `mi_decorador` es un decorador que toma una función `func` como argumento, define una nueva función `nueva_funcion` que envuelve a `func`, y luego devuelve `nueva_funcion`.

Uso del Decorador:

```python
@mi_decorador
def saludar():
    print("Hola!")
```

La línea `@mi_decorador` aplica el decorador `mi_decorador` a la función `saludar`.

Esto es equivalente a hacer:

```python
saludar = mi_decorador(saludar)
```

Llamada a la Función Decorada:

```python
saludar()
```

Al llamar a `saludar()`, se ejecuta `nueva_funcion` dentro del decorador, que imprime los mensajes antes y después de ejecutar la función original `saludar`.