1. ¿Para qué usamos Clases en Python?

Una clase es una "plantilla" a partir de la cual definimos un objeto; la clase sirve para describir cómo será y qué podrá hacer ese objeto. Imaginemos las clases como los planos para crear casas: primero necesitaríamos definir las diferentes partes de la casa (salón, cocina, dormitorios, etc.) y cómo se relacionan entre sí. Con las clases en Python sucede algo similar:

1. Definimos un **Modelo**:

    - Las clases te permiten definir un modelo para un tipo de objeto (digamos una casa, un coche, una persona...)
    - El modelo especifica las características (**atributos**) que tienen todos los objetos de ese tipo (puede ser el número de habitaciones si es una casa, la marca si hablamos de un coche…)
    - También define los comportamientos (**método**s) que pueden realizar esos objetos (abrir puertas, conducir...)

2. Crear **Objetos** a partir del Modelo:

    - Una vez que tienes el Modelo (la clase), puedes crear objetos individuales (**instancias**) a partir de él.
    - Cada objeto tendrá las mismas características y comportamientos definidos en la clase, pero pueden tener valores diferentes para cada atributo.
    - Por ejemplo, podemos crear dos instancias de la clase "Casa" con diferentes números de habitaciones, tamaño, colores…

3. Organizar y reutilizar código:

    - Las clases nos ayudan a organizar el código de manera más eficiente y modular.
    - Al agrupar características y comportamientos relacionados en una sola clase, evitamos la duplicación de código y hacemos que los programas sean más fáciles de entender y mantener.
    - Se pueden reutilizar las clases para crear diferentes tipos de objetos relacionados entre sí.

Ejemplo:

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

    def saludar (self):
        print (f'Hola, me llamo {self.nombre} {self.apellido}')

persona1 = Persona ('Ana', 'Pinon')
persona2 = Persona ('Jon', 'Arce')

persona1.saludar()
persona2.saludar()
```

- La clase "Persona" define dos atributos (características): nombre y apellido.
- También define un método (comportamiento, acción…), que en este caso sería: saludar ()
- Se crean dos objetos (instancias) de la clase Persona: persona1 y persona2.
- Se llama al método saludar () en cada objeto para imprimir un mensaje de saludo.

El resultado sería:

Hola, me llamo Ana Pinon

Hola, me llamo Jon Arce

Los beneficios de usar clases en Python:

- **Código más organizado y modular**, ya que te permite dividir el código en bloques lógicos y reutilizables.
- El código será más **fácil de mantener**, ya que los cambios en una clase se reflejan automáticamente en todos los objetos creados a partir de esa clase.
- **Reutilización de código**, puedes crear clases genéricas que se pueden usar para diferentes propósitos.
- **Flexibilidad**, puedes crear objetos con diferentes características y comportamientos.

En resumen, las clases en Python son herramientas fundamentales para crear programas estructurados, reutilizables y fáciles del mantener.

2. ¿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 se llama **constructor,** en lenguaje Python**__init__()**

Siguiendo con el ejemplo anterior, si imaginamos que una clase es como un plano para construir una casa, el constructor sería el equipo de construcción que sigue las instrucciones del plano para levantar la casa desde cero.

¿Qué hace el constructor?

- **Inicializa los atributos de la clase**: los atributos (características) son como las diferentes partes de la casa, como las paredes, ventanas o las puertas. El constructor les da un valor inicial para que la casa esté lista para ser utilizada.

- **Prepara el objeto para su uso**: El constructor puede realizar otras tareas para que el objeto esté listo para su uso, como, por ejemplo, conectar a una base de datos o abrir un archivo.

¿Cómo se ejecuta el constructor?

El constructor se ejecuta automáticamente **cada vez que se crea una nueva instancia de clase.** Esto significa que no hay que llamarlo explícitamente.

Ejemplo:

```
class Casa:
    def __init__(self, num_habitaciones, num_baños):
        self.num_habitaciones = num_habitaciones
        self.num_baños = num_baños

# Creamos una instancia de la clase casa

casa_nueva= Casa (3,2)

#Imprimimos el número de habitaciones de la casa nueva

print(casa_nueva.num_habitaciones)

# Salida:3
```

En el ejemplo, el constructor **__init__() se** ejecuta automáticamente cuando se crea la nueva instancia de la clase Casa llamada casa_nueva. El constructor inicializa dos atributos: num_habitaciones y num_baños.

A destacar:

- **Un constructor no tiene valor de retorno**: A diferencia de otros métodos, un constructor no devuelve ningún valor.
- **Una clase puede tener varios constructores**: Si se necesita inicializar los objetos de la clase de diferentes maneras se pueden definir varios constructores con diferentes parámetros.
- **No es obligatorio definir un constructor:** Si no definimos un constructor explícitamente, Python creará uno por defecto que no hace nada.

En resumen, el constructor en Python es fundamental para inicializar objetos y prepararlos para su uso. Nos permite establecer valores iniciales y realizar configuraciones necesarias de manera automática cada vez que se crea una nueva instancia de la clase.

3. ¿Cuáles son los tres verbos de API?

En el mundo de las APIs (Interfaz de Programación de Aplicaciones), hay varios verbos HTTP que se utilizan para realizar diferentes acciones sobre los recursos. Entre ellos, tres verbos son los más comunes: GET, POST y PUT.

1. **GET**: Este verbo se utiliza para obtener información de un recurso específico. Es como pedir a un camarero que nos traiga un plato de un menú. No modifica ni crea datos, simplemente los recupera.

   Ejemplos:

   - Si usamos una API para consultar el clima, un comando GET te devolvería la información meteorológica actual para una ciudad específica.
   - En una API de biblioteca, un comando GET podría devolver la lista de libros disponibles.
   - En una API de redes sociales, un comando GET podría devolver el perfil de un usuario específico.

2. **POST**: Este verbo se utiliza para crear nuevos datos en un recurso. Es como pedirle al camarero que prepare un plato especial que no está en el menú. POST envía datos al servidor para crear un nuevo recurso.

   Ejemplos:

   - Si usamos una API para gestionar tareas, un comando POST nos permitirá crear una nueva tarea con su título, descripción y fecha límite.
   - En una API de e-commerce, un comando POST podría crear un nuevo producto en el catálogo.

    - En una API de gestión de usuarios, un comando POST podría registrar un nuevo usuario con su información personal.

3. **PUT**: Este verbo se utiliza para actualizar datos en un recurso existente. Es como pedirle al camarero que nos cambie un ingrediente del plato por otro. PUT envía datos al servidor para actualizar un recurso existente completamente.

   Ejemplos:

    - Si usamos una API para gestionar una agenda de contactos, un comando PUT nos permitiría actualizar el número de teléfono de un contacto existente.
    - En una API de blogs, un comando PUT podría actualizar el contenido de una entrada específica.
    - En una API de gestión de inventarios, un comando PUT podría actualizar la cantidad de un producto específico.

Resumen:

- **GET**: Leer datos.
- **POST**: Crear datos.
- **PUT**: Actualizar datos.

Puntos importantes a tener en cuenta:

- Los verbos HTTP se envían en las peticiones a las APIs.
- El verbo HTTP que se utiliza determina la acción que se quiere realizar sobre el recurso.
- Cada API puede tener sus propias reglas para el uso de los verbos HTTP, y pueden implementarlos de manera ligeramente diferente.

Además de estos tres verbos, existen otros verbos HTTP como DELETE (para eliminar un recurso) y PATCH (para actualizar parcialmente un recurso), que también son importantes en la manipulación de datos a través de APIs.

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

MongoDB es una base de datos **NoSQL.**

Las bases de datos se clasifican en dos categorías principales:

1. Bases de datos relacionales (SQL):

- Almacenan datos en tablas con filas y columnas.
- Utilizan el lenguaje de consulta SQL para acceder y manipular datos.
- Son buenas para almacenar datos estructurados y relaciones entre ellos.
- Ejemplos: MySQL, PostgreSQL, Oracle

2. Bases de datos NoSQL:

- No tienen un esquema fijo, lo que significa que los datos pueden almacenarse en diferentes formatos.
- Utilizan diferentes lenguajes de consulta según el tipo de base de datos NoSQL.

- Son buenas para almacenar grandes cantidades de datos no estructurados o semiestructurados.
- Ejemplos: MongoDB, Cassandra, CouchDB

¿Por qué MongoDB es NoSQL?

- **Esquema flexible:** MongoDB no requiere un esquema predefinido para los datos. Esto significa que puedes almacenar datos en diferentes formatos, como JSON, BSON o incluso documentos binarios.
- **Modelo de datos de documentos:** Los datos en MongoDB se almacenan en documentos, que son estructuras JSON similares a objetos. Esto facilita el almacenamiento de datos jerárquicos y complejos.
- **Escalabilidad horizontal:** MongoDB puede escalarse horizontalmente agregando más servidores, lo que la hace ideal para aplicaciones que manejan grandes cantidades de datos.

En resumen:

Si necesitas almacenar datos estructurados con relaciones entre ellos, una base de datos SQL como MySQL o PostgreSQL podría ser una mejor opción.

Si necesitas almacenar grandes cantidades de datos no estructurados o semiestructurados, o si necesitas una base de datos que pueda escalarse horizontalmente, MongoDB podría ser una mejor opción.

5. ¿Qué es una API?

Siguiendo con ejemplos sencillos, imagina que estás en un restaurante y quieres pedir comida. No puedes ir directamente a la cocina y preparar el plato tú mismo, sino que debes comunicarte con el camarero, quien actúa como intermediario entre tú y la cocina.

Las APIs (Interfaz de Programación de Aplicaciones) funcionan de manera similar.

- **Son como camareros en el mundo digital:** Permiten que dos aplicaciones se comuniquen entre sí para intercambiar información o realizar acciones.
- **No necesitas saber cómo funciona la otra aplicación:** La API te proporciona un conjunto de instrucciones (como un menú) que puedes usar para comunicarte con ella.
- **Las APIs se utilizan en todas partes:** Están detrás de muchas de las cosas que haces online, como consultar el clima, ver videos en YouTube o iniciar sesión en una red social.

Como ejemplo, supongamos que necesitamos sabe el tiempo en Ermua. Podemos usar una API del tiempo como Euskalmet:

1. Envías una solicitud a la API (como pedirle al camarero que te traiga el menú del día).
2. La API accede a sus datos meteorológicos (como el cocinero preparando tu comida).
3. La API te devuelve la información del clima actual en Ermua (como el camarero sirviéndote el plato).

Las APIs son como intermediarios que permiten que las aplicaciones se comuniquen entre sí. Nos permiten acceder a datos y funcionalidades de otras aplicaciones sin necesidad de saber cómo funcionan.

Se utilizan en una amplia variedad de aplicaciones, desde el clima hasta las redes sociales.

6. ¿Qué es Postman?

Podemos ponernos en el lugar de alguien que llega por primera vez a un país que no conoce, no habla el idioma y no sabe cómo funcionan sus servicios. Necesitaría una herramienta que le ayudase a traducir los mensajes y a entender sus respuestas o un intérprete que hiciese de intermediario.

¡Aquí es donde entra en juego Postman!

Postman es como un traductor universal para APIs.

- Te permite enviar solicitudes a las APIs y ver sus respuestas.
- Puedes probar diferentes métodos HTTP (como GET, POST, PUT y DELETE) para interactuar con las APIs.
- Puedes ver los encabezados, el cuerpo y las cookies de las solicitudes y respuestas.
- Puedes guardar tus solicitudes y respuestas como "colecciones" para reutilizarlas más tarde.
- Puedes colaborar con otros desarrolladores en las colecciones de Postman.

Postman es una herramienta esencial para cualquier desarrollador que trabaje con APIs., ya que te permite diseñar, crear, probar y documentar APIs, te ayuda a ahorrar tiempo, a evitar errores y a asegurarte de que tus aplicaciones se comunican correctamente.

Ejemplo:

Imagina que estás desarrollando una aplicación que necesita acceder a la API de Twitter para obtener los últimos tweets de un usuario específico.

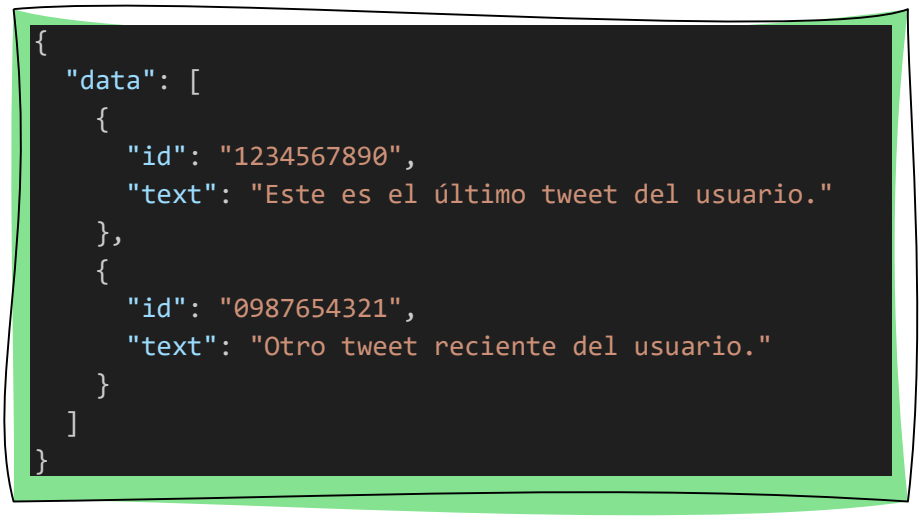
1. **Creación de la solicitud GET:**
   Usas Postman para crear una solicitud GET a la API de Twitter.

2. **Especificiación de la URL y parámetros:**
   Especificas la URL de la API y el usuario de Twitter del que quieres obtener los tweets. Por ejemplo, la URL podría ser

   https://api.twitter.com/2/tweets?username=example_user.

3. **Envío de la solicitud:**
   Envías la solicitud y Postman te muestra la respuesta de la API.

4. **Visualización de la respuesta:**
   La respuesta contiene los últimos tweets del usuario, en formato JSON (un formato de texto para el intercambio de datos). Aquí tienes un ejemplo de cómo podría verse la respuesta:



```
{
  "data": [
    {
      "id": "1234567890",
      "text": "Este es el último tweet del usuario."
    },
    {
      "id": "0987654321",
      "text": "Otro tweet reciente del usuario."
    }
  ]
}
```

5. **Uso de los datos:**
   Puedes usar los datos de la respuesta para mostrar los tweets en tu aplicación.

En resumen, Postman no solo actúa como un intérprete entre el desarrollador y las APIs, sino que también proporciona un entorno robusto y amigable para la gestión y prueba de servicios web, lo que lo convierte en una herramienta indispensable en el desarrollo de software.

7. ¿Qué es el polimorfismo?

El polimorfismo es una característica de la programación orientada a objetos (como es Python) que permite que un mismo objeto pueda responder a diferentes métodos de forma distinta. En términos simples, es como si una misma persona pudiera realizar diferentes roles en diferentes situaciones.

¿Cómo funciona el polimorfismo?

El polimorfismo se basa en el concepto de herencia, donde:

- Las clases pueden heredar propiedades y métodos de otras clases.
- Las clases hijas pueden "heredar" el comportamiento de sus clases padres.
- Las clases hijas también pueden reescribir (sobrescribir) los métodos heredados para que se comporten de manera diferente.

Ejemplo:

```
class Vehiculo:

    def mover(self):
        print("El vehículo se mueve")

class Coche(Vehiculo):
    def mover(self):
        print("En el coche se conduce")

class Bicicleta(Vehiculo):
    def mover(self):
        print("En la bicicleta se pedalea")

class Avion(Vehiculo):
    def mover(self):
        print("El avión vuela")

# Crear instancias de las clases
coche = Coche()
bicicleta = Bicicleta()
avion = Avion()

# Llamar al método mover() en cada instancia
coche.mover()      # Salida: En el coche se conduce
bicicleta.mover()  # Salida: En la bicicleta se pedalea
avion.mover()      # Salida: El avión vuela
```

Imaginemos que tenemos una clase base llamada "vehículo" con un método llamado "mover()".

- La clase "vehículo" podría tener una implementación genérica del método "mover ()", por ejemplo, imprimir "El vehículo se mueve".

- Ahora podemos crear clases hijas de “vehículo” como “coche”, “bicicleta” y “avión”.
- Cada una de estas clases hijas puede sobrescribir el método “mover ()” para que se adapte a su forma de movimiento: conducir, pedalear y volar, respectivamente.

En resumen, el **polimorfismo** es una característica de la programación orientada a objetos que permite que un mismo objeto pueda responder a diferentes métodos de forma distinta. **Se basa en la herencia**, donde las clases hijas pueden heredar y sobrescribir métodos de sus clases padres. **Beneficios:** Facilita la escritura de código flexible, reutilizable y escalable, haciendo que las aplicaciones sean más potentes y fáciles de mantener.

8. ¿Qué es un método dunder?

En el mundo de la programación con Python, los métodos dunder, también conocidos como métodos mágicos o métodos especiales, son aquellos que tienen un doble guion bajo (‘__’) al principio y al final de su nombre. Estos métodos permiten personalizar y extender el comportamiento de las clases y objetos.

¿Para qué sirven los métodos dunder?

- **Personalizar el comportamiento de las clases y objetos:** Permiten definir cómo se comportan tus clases y objetos en diferentes situaciones, como al imprimirse, compararse o convertirse en cadenas de texto.
- **Simplificar tu código:** Ayudan a evitar código repetitivo, haciendo que tu código sea más legible y fácil de entender.
- **Agregar funcionalidades avanzadas:** Permiten implementar funcionalidades que no son posibles con las herramientas de programación estándar.

Algunos ejemplos de métodos dunder:

- **init(self):** Este método se llama automáticamente cuando se crea una nueva instancia de una clase. Se utiliza para inicializar los atributos del objeto.
- **str(self):** Este método se llama cuando se quiere convertir un objeto en una cadena de texto. Devuelve una representación textual del objeto.
- **repr(self):** Este método es similar a __str__(self), pero se utiliza para una representación más formal del objeto, a menudo utilizada para depuración.
- **add(self, other):** Este método se llama cuando se usan los operadores + o += con un objeto. Define cómo se deben sumar dos objetos.
- **eq(self, other):** Este método se llama cuando se usa el operador == para comparar dos objetos. Define si dos objetos son iguales.

Ejemplo:

```
class Persona:

    def __init__(self, nombre, apellido):
        self.nombre = nombre
        elf.apellido = apellido

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

    def __add__(self, other):
        return f"{self.nombre} {other.apellido}"

persona1 = Persona("Ana", "Pinon")
persona2 = Persona("Jon", "Arce")

print(persona1)  # Salida: Ana Pinon
print(persona1 + persona2)  # Salida: Ana Arce
```

En este ejemplo, hemos definido:

- **`__init__`** para inicializar los objetos **`Persona`.**
- **`__str__`** para definir cómo se convierte una instancia de **`Persona`** en una cadena de texto.
- **`__add__`** para definir cómo se comporta el operador **`+`** cuando se usa con instancias de **`Persona`.**

En resumen, los métodos dunder son métodos especiales en Python que te permiten personalizar el comportamiento de las clases y objetos a la vez que te ayudan a escribir código más flexible, reutilizable y potente. Existen muchos métodos dunder diferentes, cada uno con un propósito específico.

9. ¿Qué es un decorador en Python?

Un decorador en Python son funciones especiales que permiten modificar el comportamiento de otras funciones sin tener que cambiar su código original.

¿Cómo funcionan los decoradores de Python?

- La nueva función decorada tendrá el mismo nombre y firma que la función original, pero su comportamiento interno puede ser diferente.
- El decorador puede agregar código ante, después o en lugar de la ejecución de la función original.

Ejemplo:

```
# 1. Definimos el Decorador
def mi_decorador(func):
    def envoltura():
        print ('Algo se va a ejecutar...')
        func()
        print ('Algo ha terminado de ejecutarse')
    return envoltura

# 2. Definimos la función que queremos decorar

def di_hola():
    print ('Hola')

# 3. Aplicamos el decorador
@mi_decorador
def di_hola():
    print ('Hola')

di_hola()

#salida:
#Algo se va a ejecutar...
#Hola
#Algo ha terminado de ejecutarse
#Adios
```

1. **Definimos el Decorador**:

- '**mi_decorador**' es una función que toma otra función ('**func**') como argumento.
- Dentro de '**mi_decorador**', se define una función '**envoltura**' que envuelve la llamada a '**func**' con código adicional.
- '**mi_decorador**' devuelve la función '**envoltura**'

2. **Aplicación del Decorador:**

- Cuando aplicamos '**mi_decorador**' a '**di_hola**', se crea una nueva función ('**envoltura**') que extiende la funcionalidad de '**di_hola**' sin modificarla directamente.
- Usando '**@mi_decorador**' encima de '**di_hola**' , le decimos a Python que queremos aplicar '**mi_decorador**' a '**di_hola**'

3. **Ejecución**

- Al llamar a '**di_hola()**', en realidad estamos llamando a '**envoltura()**' que incluye la funcionalidad adicional ante y después de la llamada a **'di_hola'**.

Beneficios de usar decoradores de Python:

- **Código más limpio y organizado:** Te permiten separar la lógica de la decoración del código de la función original.
- **Código más flexible:** Te permiten modificar el comportamiento de las funciones sin tener que cambiar su código original.
- **Código más reutilizable:** Puedes crear decoradores genéricos que se pueden aplicar a diferentes funciones.

En resumen, un decorador en Python sería, de forma muy sucinta, una función que envuelve a otra función, y se usa para añadir comportamiento adicional a esas funciones de manera limpia y ordenada.