1. ¿Para qué usamos Clases en Python?

En Python una clase es una plantilla que nos permite crear objetos, también llamados instancias. La clase permite definir las propiedades y el comportamiento común a todos los objetos que generemos a partir de esa clase. Asimismo, cabe decir que las clases pueden albergar dentro de sí mismas tanto datos como funciones.

La **sintaxis básica** de una clase es la siguiente:

```
class Empresa:
    def saludo (self):
        return 'Hola'
```

- La convención determina que el nombre de la clase debe iniciarse con letra mayúscula.
- Si observamos la función 'saludo()' podemos identificar que contiene el argumento 'self', es un argumento que se pone por defecto en los métodos (funciones) que se encuentran dentro de las clases.

Para llamar a la clase tenemos que crear una instancia (objeto):

- La instancia hace referencia a un objeto específico creado a partir de una clase.
- Cuando definimos una clase estamos creando, por así decirlo, una especie de molde para crear objetos.
- La instancia sería ese objeto que creamos utilizando la plantilla/molde que nos ofrece la clase.

  ```
  facebook = Empresa()
  ```

  Aquí estamos creando un objeto llamado 'facebook' a partir de la clase empresa.

Una vez creado ese objeto 'facebook' podemos aplicar los métodos que contiene la clase en este caso 'saludo()' de la siguiente manera:

```
print(facebook.saludo()) #Salida del terminal --> Hola
```

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

El método que se ejecuta de manera automática es '_ _ init _ _' (pronunciado dunder init). Se caracteriza por ser un método especial ya que se utiliza para inicializar objetos cuando se crea una instancia de una clase.

Cuando definimos una clase en Python, este método se llama de manera automática cuando está presente dentro de la clase y se utiliza para establecer los atributos del objeto.

Tomando como referencia el ejemplo anterior, **la sintaxis para su aplicación** sería la siguiente:

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

    def saludo (self):
         return f'Hola, nuestra empresa se llama {self.nombre} y cuenta
con {self.empleados} empleados'
```

Mediante el método '__init__' estamos asignado a la instancia dos atributos, indicando que el atributo nombre y empleados sean igual a las variables nombre y empleados.

Creamos el objeto 'facebook' de nuevo, pero esta vez le introduciremos dos valores a la instancia, los correspondientes a 'nombre' y 'empleados':

```
facebook = Empresa ('Facebook', 76480)
```

Imprimimos el objeto 'facebook' aplicando el método 'saludo()' para ver cómo se comporta:

```
print(facebook.saludo())  #Salida del terminal --> Hola, nuestra
empresa se llama Facebook y cuenta con 76480
```

El método 'saludo()' de la clase Empresa devuelve una cadena formateada que incluye el nombre y la cantidad de empleados de la empresa, utilizando los valores proporcionados al crear la instancia del objeto 'facebook'.

3. ¿Cuáles son los tres verbos de API?

3.1. POST:

- Se utiliza para enviar datos nuevos al servidor para su procesamiento o almacenamiento.

- Ejemplo de aplicación sin considerar la configuración de la base de datos, la definición del modelo de la base de datos ni el esquema de serialización:

```
@app.route('/guía', methods = ['POST'])
def añadir_guía():
    título = request.json['título']
    contenido = request.json['contenido']

    nueva_guía = Guía(título, contenido)

    db.session.add(nueva_guía)
    db.session.commit()

    guía = guía.query.get(nueva_guía.id)

    return guía_schema.jsonify(guía)
```

1) **@app.route('/guía', methods=['POST'])** → Este decorador de ruta indica que esta función maneja las solicitudes POST enviadas a la URL '/guía'.

2) **def añadir_guía()** → Esta es la función que maneja las solicitudes POST. Se ejecuta cuando se envía una solicitud POST a la URL '/guía'.

3) **título = request.json['título']** → Extrae el título de la guía del cuerpo JSON de la solicitud POST.

4) **contenido = request.json['contenido']** → Extrae el contenido de la guía del cuerpo JSON de la solicitud POST.

5) **nueva_guía = Guía(título, contenido)** → Crea un nuevo objeto de la clase Guía utilizando el título y el contenido extraídos de la solicitud.

6) **db.session.add(nueva_guía)** → Agrega la nueva guía a la sesión de la base de datos.

7) **db.session.commit()** → Confirma los cambios en la sesión de la base de datos, lo que efectivamente guarda la nueva guía en la base de datos.

8) **guía = Guía.query.get(nueva_guía.id)** → Recupera la guía recién creada de la base de datos utilizando su ID.

9) **return guía_schema.jsonify(guía)** → Devuelve la representación JSON de la guía recién creada como respuesta a la solicitud POST.

3.2. GET:

- Se utiliza para recuperar datos del servidor.

- Ejemplo de aplicación bajo las mismas circunstancias que el anterior.

```
@app.route('/guías', methods = ['GET'])
def get_guías():
    todas_guías = Guía.query.all()
    resultado = guías.schema.dump (todas_guías)
    return jsonify (resultado)
```

1) **@app.route('/guías', methods=['GET'])** → Define una ruta en la aplicación para manejar solicitudes GET a la ruta '/guías'.

2) **def get_guías()** → Define una función llamada get_guías() que maneja las solicitudes GET a la ruta '/guías'.

3) **todas_guías = Guía.query.all()** → Recupera todas las guías de la base de datos utilizando el método all() de la consulta guía.query.

4) **resultado = guías.schema.dump(todas_guías)** → Serializa todas las guías recuperadas en formato JSON utilizando el esquema guías.schema y el método dump().

5) **return jsonify(resultado)** → Devuelve el resultado serializado en formato JSON como respuesta a la solicitud GET utilizando la función jsonify().

3.3. PUT:

- Se utiliza para actualizar datos existentes en el servidor.

```
@app.route('/guía/ <id>', methods = ['PUT'])

def actulizar_guía(id):

    guía = Guía.query.get(id)

    título = request.json ['título']

    contenido = request.json ['contenido']



    guía.título = título

    guía.contenido = contenido



    db.session.commit()

    return guía_schema.jsonify(guía)
```

1) **@app.route('/guía/<id>', methods=['PUT'])** → Indica que la ruta /guía/<id> manejará peticiones HTTP PUT. <id> es una variable de ruta que contendrá el ID de la guía que se va a actualizar.
2) **def actualizar_guía(id):** → Esto define una función llamada actualizar_guía que maneja las solicitudes a la ruta /guía/<id>.
3) **guía = Guía.query.get(id)** → Utiliza el ID proporcionado en la URL para buscar la guía correspondiente en la base de datos.
4) **título = request.json['título']** → Extrae el título de la guía del cuerpo de la solicitud JSON.
5) **contenido = request.json['contenido']** → Extrae el contenido de la guía del cuerpo de la solicitud JSON al igual que en el título
6) **guía.título = título** → Actualiza el título de la guía con el nuevo valor proporcionado en la solicitud.
7) **guía.contenido = contenido** → Actualiza el contenido de la guía con el nuevo valor proporcionado en la solicitud.
8) **db.session.commit()** → Guarda los cambios realizados en la guía en la base de datos.

9) **return guía_schema.jsonify(guía)**: Devuelve la guía actualizada como una respuesta JSON.

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

MongoDB es una base de datos NoSQL, esto quiere decir que son bases de datos a una que no siguen el modelo relacional utilizado en las bases de datos SQL tradicionales. En el caso de MongoDB el modelo de datos que utiliza es de documentos.

En este modelo, los datos se representan y almacenan en forma de documentos similares a JSON. Cada documento es una entidad de datos independiente que puede contener varios campos con valores de diferentes tipos de datos, incluidos otros documentos anidados y matrices. Este enfoque permite una gran flexibilidad en la estructura de los datos.

Sin embargo, hay que tener cuidado con esa flexibilidad que nos permite MongoDB, ya que un abuso de ella puede conllevar implicaciones en el diseño de la base de datos y en el rendimiento de las consultas.

Por ejemplo, imaginemos que tenemos una colección llamada 'Empleados' en la que almacenamos diferentes documentos que representan a la información de cada empleado que forma nuestra empresa.

1º) Documento → Empleado1

```
{
    "nombre": "Jon Madariaga",
    "cargo": "Desarrollador",
    "conocimientos": ["Python", "JavaScript", "MongoDB"],
}
```

2º) Documento → Empleado2

```
{
    "nombre": "Valeria Vallejo",
    "cargo": "Diseñador",
    "herramientas": ["Photoshop", "Illustrator"],
}
```

Imaginemos que queremos realizar una consulta sobre el nombre y los conocimientos que tienen nuestros empleados. Realizaríamos la siguiente consulta:

```
db.empleados.find({}, {'nombre':1 ,'conocimientos': 1)
```

El primer documento, sí nos saldría nombre y experiencia asociados, pero en el segundo tan sólo nos devolvería el nombre, ya que el campo conocimientos no se encuentra dentro del documento, sino que se ha denominado herramientas.

En resumen, este ejemplo sintétiza que a pesar de la flexibilidad es importante mantener cierta consistencia entre los campos de los documentos, ya que luego puede resultar costoso realizar las consultas a la base de datos.

5. ¿Qué es una API?

Una API define cómo se pueden comunicar dos sistemas diferentes. Proporciona una forma estructurada y estandarizada para que las aplicaciones interactúen entre sí, permitiendo que las aplicaciones puedan compartir datos, funciones y servicios.

6. ¿Qué es Postman?

Postman es una herramienta de colaboración para el desarrollo de APIs que permite a los desarrolladores crear, probar, compartir, documentar y monitorear APIs de manera fácil y eficiente.

7. ¿Qué es el polimorfismo?

El polimorfismo es un principio de la programación orientado a objetos que permite que objetos de diferentes clases sean tratados de manera uniforme si cumplen con el requisito de compartir métodos entre clases, es decir, a través de la sobrescritura de métodos

Ejemplo básico:

```
class ElementoHTML:

    def __init__(self, contenido):

        self.contenido = contenido



class Encabezado(ElementoHTML):

    def generar_html(self):

        return f"<h1>{self.contenido}</h1>"



class Parrafo(ElementoHTML):

    def generar_html(self):

        return f"<p>{self.contenido}</p>"



class Lista(ElementoHTML):

    def generar_html(self):

                items_html = ''.join(f"<li>{item}</li>" for item in
self.contenido)

        return f"<ul>{items_html}</ul>"



# Crear instancias de diferentes elementos HTML

encabezado = Encabezado("Título")

parrafo = Parrafo("Este es un párrafo de ejemplo.")

lista = Lista(["Item 1", "Item 2", "Item 3"])
```

```
print(encabezado.generar_html())#Salida: <h1>Título</h1>

print(parrafo.generar_html())#Salida:<p>Este es un párrafo de
ejemplo.</p>

print(lista.generar_html())#Salida:<ul><li>Item1</li><li>Item2</li><li
>Item 3</li></ul>
```

En objetos de diferentes clases podemos aplicar 'el mismo método' ya que todas las clases comparten el método 'generar_html()'.

8. ¿Qué es un método dunder?

Un método dunder es un método que se implementa en una clase, permitiendo personalizar el comportamiento predeterminado de los objetos de una clase. Algunos ejemplos de los dunder methods son:

1) **'__init__':** se llama automáticamente cuando se crea una nueva instancia de la clase y se utiliza para inicializar los atributos de la clase.

```
class Persona:

    def __init__(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad
```

2) **'__str__':** se llama automáticamente cuando se utiliza la función str() en un objeto y se utiliza para devolver una representación legible de una cadena del objeto.

```
class Persona:

    def __init__(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad



    def __str__(self):

        return f'Nombre: {self.nombre}, Edad: {self.edad}'
```

```
#Instancia

persona1 = Persona('Juan', 30)

print(str(persona1)) #Salida --> Nombre: Juan, Edad: 30
```

3) **'__repr__':** proporciona una representación 'oficial' del objeto. A diferencia del método '__str__' que se utiliza para proporcionar una representación legible de la cadena.

```
class Persona:

    def __init__(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad



    def __repr__(self):

        return f'Nombre: {self.nombre}, Edad: {self.edad}'

#Instancia

persona1 = Persona('Juan', 30)

print(repr(persona1)) #Salida --> Nombre: Juan, Edad: 30
```

*** Cabe matizar que no hace falta llamar explícitamente a la función 'repr()' o 'str()' para que esta se ejecute, mediante print(objeto) sería suficiente. Pero en caso de que una clase contenga un método '__str__' y '__repr__' prevalecerá la impresión de '__repr__'.***

9. ¿Qué es un decorador de python?

El decorador @property en Python se utiliza para convertir un método de una clase en una propiedad, lo que permite acceder a ese método como si fuera un atributo simple, sin la necesidad de tener que llamar al método explícitamente.

Su utilidad reside en tener la capacidad de poder controlar los datos, sin permitir el acceso directo a los atributos subyacentes de la clase.

Ejemplo básico:

```
class Persona:

    def __init__(self,nombre):

         self._nombre = nombre # '_' es la convención para señalar que
el atributo está protegido



    @property

    def nombre (self):

        return self._nombre



#Instancia

persona1 = Persona('Jon')



#Imprimimos el objeto

print(persona1.nombre) #Salida -> Jon
```