1. ¿Para qué usamos Clases en Python?

Qué es una clase

Una clase es como una plantilla que define las características (**atributos**) y los comportamientos (**métodos**) que tendrán los objetos que se creen a partir de ella.
Las clases son como ***"mapeadores de objetos"***, que te permiten construir una estructura o esquema para los objetos. Por lo tanto, contienen **datos, funciones, y también comportamiento**.

Utilidad de las clases

Las clases ayudan a estructurar el código de una forma limpia y ordenada, permitiendo la **reutilización** de su código sin el arduo trabajo de tener que codificar de nuevo.

Ejemplo:

```
class Almacen:
   proposito = 'almacenaje'
   zona = 'oeste'

a1 = Almacen()                #esta es una instancia concreta

print(a1.proposito, a1.zona)    # devuelve:  almacenaje oeste
```

Qué es una instancia

Una instancia es una **realización concreta de esa clase** o plantilla, un objeto real con valores específicos para sus atributos. Al crear una nueva clase, se crea un **nuevo** tipo de **objeto**, permitiendo crear **nuevas instancias** de ese tipo.

Cada **instancia de clase** puede tener **atributos adjuntos** para mantener su estado. Las instancias de clase también pueden tener **métodos** (definidos por su clase) para modificar su estado.

En este punto, definamos dos tipos de atributo.

Dos tipos de atributos:

- **Atributos de instancia**: Pertenecen a la **instancia de la clase o al objeto**. Son atributos particulares de cada instancia.
- **Atributos de clase**: Se trata de atributos que pertenecen **a la clase**, por lo tanto serán **comunes** para todos los objetos.

Sintaxis:

La sintaxis básica para definir una clase en Python es: **class NombreDeLaClase:** seguida de un bloque de código con las definiciones de sus atributos y métodos.

---

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

El método que se ejecuta **automáticamente** por el intérprete de Python (sin ser llamado) cuando se crea una instancia de una clase es el método **constructor** (también llamado **inicializador** o método **__init__** en Python). Este método, más que 'construir' en realidad se encarga de **inicializar los atributos de la clase** y realizar cualquier acción necesaria para preparar el objeto para su uso. El método **__init__** . Es uno de los métodos con la denominación en inglés de **Dunder** de Python (y que explicaremos con detalle en **el punto nº 8** de estos ejercicios), llamados así porque comienzan y finalizan con dos guiones bajos seguidos (___)

Ejemplo:

```
class MyClass:
    # Método constructor creado a crear el objeto:
    def __init__(self, atributo1, atributo2):
        self.atributo1 = atributo1
        self.atributo2 = atributo2

    # Método definido dentro de la clase:
    def mi_metodo(self):
    return f"Atributo nº 1: {self.attributo1}, Atributo nº 2: {self.atributo2}"
```

El argumento **(self)** se utiliza al declarar una función como método para **dar un nombre a esta variable de instancia dentro del ámbito de la función**; se trata de una referencia a la instancia actual del objeto que se está creando. El nombre **self** es una **convención estándar** para que otros codificadores (o uno mismo) sea consciente de la intención de referirse a la instancia que llama al método. El argumento **(self)** permite:

- **acceder a los atributos de la instancia** (en nuestro caso **self.atributo1** y **self.atributo2**).
- **llamar a los métodos de dicha instancia**. Los métodos de instancia son métodos que actúan sobre las instancias de una clase. Tienen acceso a los atributos de esas instancias a través del parámetro **(self)** . Por ejemplo **self.mostrar_nombre()** llama al método **mostrar_nombre** de la instancia.
- **referenciar la instancia misma**, ya que **self** representa al objeto de la clase.

Veamos otro ejemplos:

```
    class Gato:
    # El método __init__ es llamado al crear el objeto:

        def __init__(self, nombre, raza):
            print(f"Creando gato {nombre}, {raza}")

            # Atributos de instancia:
            self.nombre = nombre
            self.raza = raza
```

```
        #creando otro método de la clase:
        def maullar(self)
            print(f" {self.nombre} maulla: Miauuuu")

    #creando objeto pasando el valor de los atributos (creando instancia de la clase Gato):
    mi_gato = Gato("Sammy", "tuxido")   #devuelve: Creando gato Sammy, tuxido

    #llamando al método maullar
    mi_gato.maullar()                   #devuelve:    Sammy maulla: Miauuuu
```

Notas:

- El argumento **(self)**: cuando se llama a un método dentro de una instancia, no se necesita pasar **(self)** explícitamente como argumento
- En el método **maullar(self)**, ***(self)*** permite acceder a los **atributos "nombre"** y **"raza"** de la instancia actual, e imprime *"Sammy maulla: Miauuuu"*.
- Al llamar a **mi_gato.maullar()**, estamos utilizando una **instancia** llamada **mi_gato** como el valor de **(self)** dentro del método.

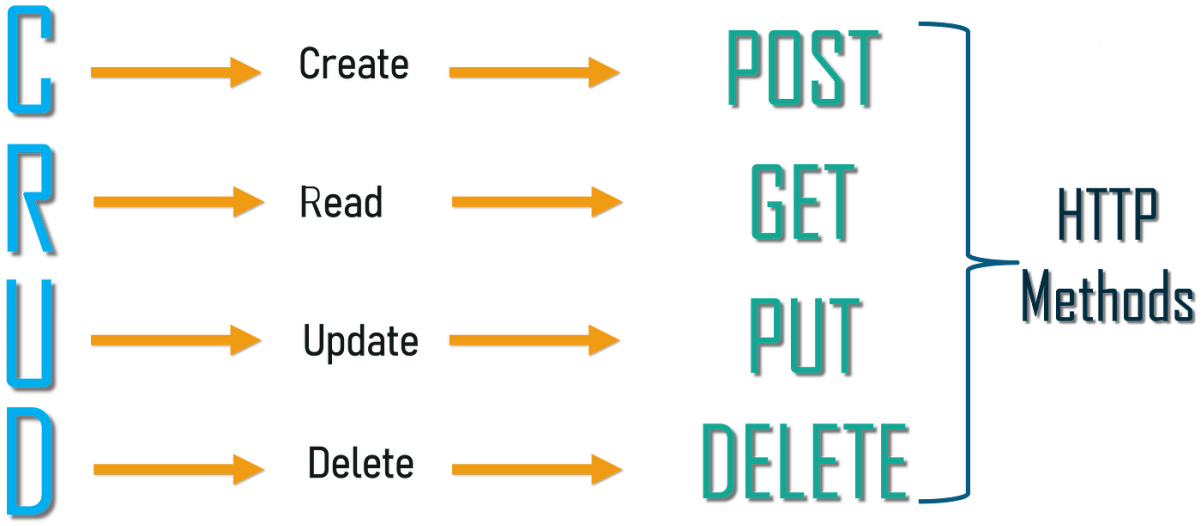
---

3. ¿Cuáles son los tres verbos de API?

En ***API Flask*** (y en general en el protocolo **HTTP**), los nombres de los tres verbos **más comunes** -y que definen cómo se interactúa con los recursos de una **API**- son los siguientes (junto con sus correspondientes operaciones):

GET	POST	PUT
Lectura	Creación	Actualización

(y para eliminar un recurso, utilizaremos **DELETE**):



Detalle de las funciones:

- **GET**:

  Este verbo se utiliza para **recuperar información** de un recurso específico. Utilizaremos este método para cualquier llamada de **API** externa dentro de ***Flask***. Ejemplos:

  - Se puede usar para obtener los datos de un usuario con un ID determinado
  - Para obtener una lista de todos los productos en una tienda online.

- **POST**: este verbo se utiliza para **crear un nuevo recurso**. Ejemplos:
  - Se puede usar para registrar un nuevo usuario en una aplicación
  - Para crear una nueva orden de compra en una tienda online.
- **PUT**:

  Este verbo se utiliza para **actualizar completamente** un recurso existente. En otras palabras, se envía **una nueva versión completa** del recurso, reemplazando la versión anterior. Ejemplo:

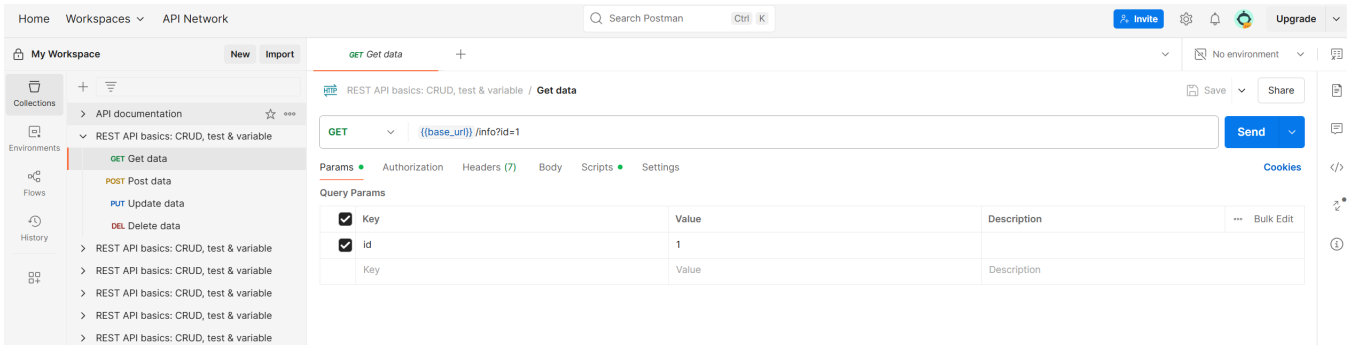
  - Se puede usar para actualizar los datos de un usuario, reemplazando todos los campos de su información.

Ejemplo:

Acción	URL (endpoint)	Verbo HTTP
Obtener publicaciones	/publicaciones	get
Obtener una publicación en específico	/publicaciones/: id	get

Acción	URL (endpoint)	Verbo HTTP
Registrar publicación	/publicaciones	post
Actualizar una publicación	/publicación/:id	put
Actualizar una publicación en específico	/publicación/ :id	put

Y aquí los tenemos en ***Postman*** (explicaremos mejor ***Postman*** en el **punto 6** de estos ejercicios) elegiríamos dentro de las opciones de la izquierda (**GET/POST/PUT/DELETE**) y añadiríamos en *endpoint* en cuestión. Un ***endpoint API*** es la "puerta de entrada", un punto de conexión a una parte específica de la **API**. Se refiere a **la dirección URL específica** de una recurso en una API (*Interfaz de Programación de Aplicaciones*). Es el punto de acceso a través del cual una aplicación o servicio interactúa con la **API** para acceder a datos o funciones:



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

- **MongoDB** es una base de datos **NoSQL** que utiliza un modelo flexible y orientado a documentos. Se basa en el teorema **CAP**, priorizando la ***disponibilidad y tolerancia*** a la partición. Almacena datos en documentos **JSON** (JavaScript Object Notation) o **BSON** (Binary JSON) dentro de colecciones, lo que ofrece **flexibilidad para modificar** el esquema de los datos.

- Una base de datos **SQL** es **relacional** y organiza los datos en tablas con **esquemas predefinidos**. Es decir, almacena datos en tablas estructuradas con filas y columnas, donde cada tabla tiene una relación definida con otras tablas. Las propiedades **ACID** son cruciales para garantizar que las transacciones en bases de datos sean **correctas, fiables y que los datos se mantengan íntegros**. Son esenciales para sistemas que manejan datos sensibles, como transacciones bancarias o sistemas de gestión de información crítica.

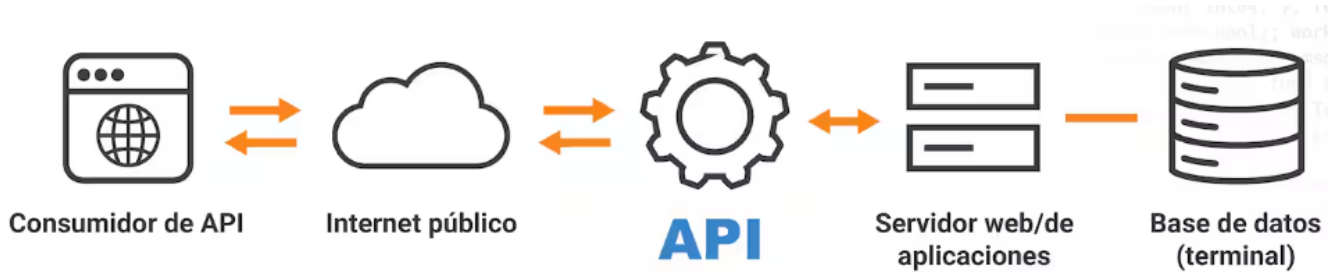
	MongoDB (NoSQL)	SQL
Enfoque	**Enfoque flexible**: almacena datos en documentos **JSON** (JavaScript Object Notation) o **BSON** (Binary JSON) dentro de **colecciones** (agrupaciones de documentos), lo que ofrece flexibilidad para modificar el esquema de los datos.	**Esquema predifinido**. Relacional y tabular: organiza datos en tablas estructuradas con **filas y columnas**, donde cada tabla tiene una relación definida con otras tablas. Cada tabla tiene una **clave principal** que se utiliza para identificarla, y la clave externa crea una relación.
Escalabilidad	**Horizontalmente**, particionando los datos en múltiples servidores para mayor disponibilidad. Dos características clave para escalar: • Conjuntos de réplicas: grupos de servidores MongoDB que contienen datos idénticos • Partición: diferentes partes de sus datos distribuidas en diferentes servidores	**Verticalmente**, añadiendo recursos a un único servidor. Replicación de lectura mediante la creación de copias de solo lectura de la base de datos en otros servidores
Caso de uso	Se adapta bien a aplicaciones que requieren almacenamiento flexible de **datos no estructurados o semiestructurados**, como aplicaciones web con contenido dinámico, aplicaciones móviles o almacenamiento de logs.	Ideal para aplicaciones que requieren **datos estructurados y transacciones complejas**, como sistemas de gestión empresarial o sistemas financieros.
Modificaciones	Los documentos de MongoDB siguen un modelo de datos jerárquico y mantienen la mayoría de los datos en un solo documento, lo que reduce la necesidad de unir varios documentos. Ofrece una **API** `insertMany()` para insertar datos rápidamente y priorizar el rendimiento de escritura.	Requiere que **los datos se inserten fila por fila**, por lo que el rendimiento de escritura es más lento. Diseñada para implementar **uniones de alto rendimiento** en varias tablas que están indexadas adecuadamente

En cuanto a CUÁL ELEGIR, dependerá absolutamente de los **objetivos** de nuestro proyecto:

- **Para transacciones complejas** (como por ejemplo ***joins*** entre tablas) y con una necesidad alta de integridad de datos deberíamos usar **SQL**.
- Sin embargo, para **datos no estructurados** o docs que puedan variar en estructura, documentos de gran tamaño, aplicaciones web/móvil para almacenar datos dinámicos, elegiríamos un sistema **NoSQL** como **MongoDB**.

---

5. ¿Qué es una API??



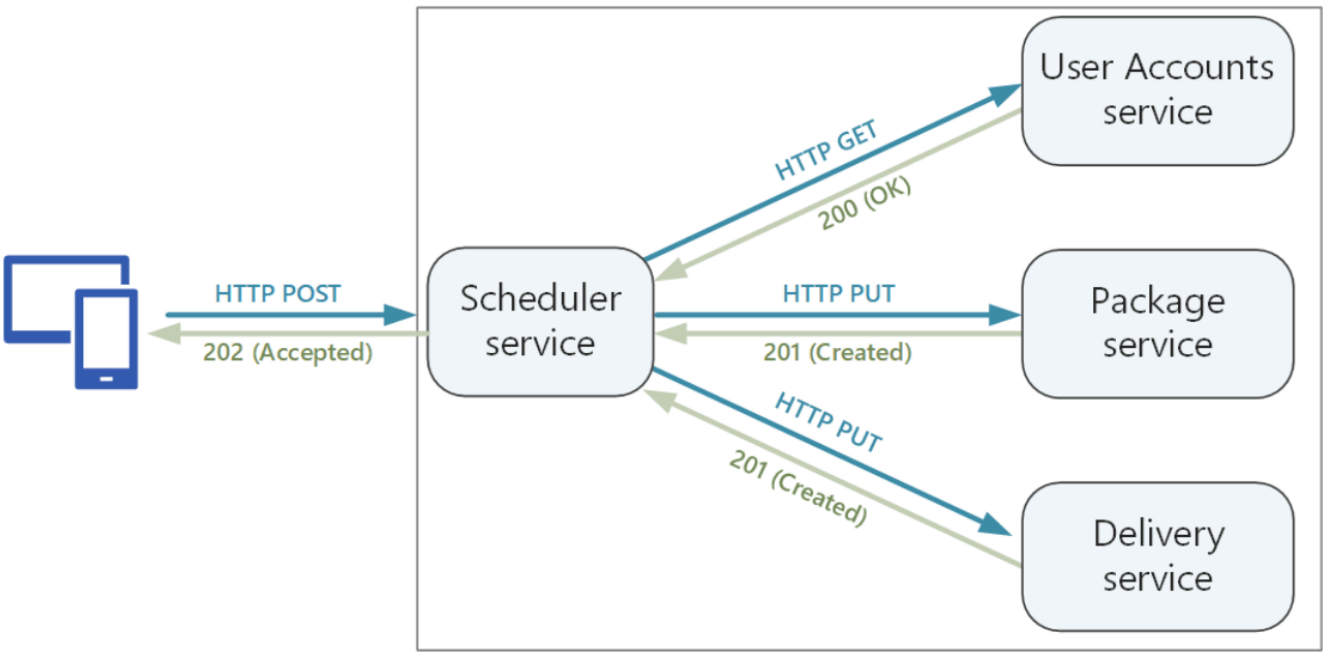
Una **interfaz de programación de aplicaciones**, o **API**, es un conjunto de protocolos y definiciones que permiten que **diferentes programas y componentes de software se comuniquen entre sí**, y compartan datos y funcionalidad; es decir, sirven para establecer **un punto de conexión o interacción entre dos sistemas software**. Sin ellas, por ejemplo, no sería posible la conexión entre redes sociales, plataformas online, sistemas operativos o bases de datos.

Dicho de forma sencilla, las **API** es una forma que podemos comunicarnos con una aplicación, sin necesidad de implementar aplicaciones tipo ***web scraper*** ("raspado Web", extraer datos de una web de forma automática). Nos da una serie de comandos para comunicarnos con otro servidor u otra aplicación. Te da una serie de **endpoints** o **URL**s. Típicamente las **API** retornan datos **JSON** (aunque también pueden manejar XML, HTML, texto...). En definitiva retornan algo que realmente pueden utilizar aplicaciones externas (**React, View, Angular...**). Si tratáramos de analizar los datos que necesitamos de la URL de una API, sería bastante caótico. Por esto existen aplicaciones como **Postman** que veremos en el siguiente punto.

Las **API** se utilizan para **desarrollar e integrar el software de las aplicaciones**. A diferencia de **una interfaz de usuario** (**UX**), que conecta a una persona con un ordenador, una **API** conecta a **dos softwares** o partes de un software. Cuando un usuario utiliza una aplicación o web, no tendrá acceso a la **API** , aunque disfrute de sus ventajas.

Las **API** son esenciales para:

- los **servicios en la nube** *(p.ej. Amazon S3)*
- los **microservicios** *(p.ej. una plataforma de comercio electrónico)*
- las **arquitecturas sin servidor** (por su escalabilidad, flexibilidad y fácil mantenimiento)
- el **Internet de las cosas (IoT)** *(p.ej. hogar inteligente -termostatos,luces, cerradoras, sensores médicos, vehículos autónomos)* de los que dependen muchos entornos de TI.



Estructura de una API para MICROSERVICIOS

Las **API** también rigen cómo se **permite que interactúen las aplicaciones**, y controlan **cómo** se realizan las **solicitudes** y los **tipos de solicitudes** que se pueden intercambiar entre programas.

Importancia de la Seguridad:

- Los puntos de conexión de las **API** hacen que el sistema sea **vulnerable a los ataques**. La supervisión de las **API** es crucial para evitar su uso indebido.
- Dado que exponen la **lógica y los recursos** de las aplicaciones, y a menudo implican la transferencia de **información confidencial**, las **API** son un objetivo atractivo para los **hackers**. Una **API no protegida** puede hacer posible que agentes maliciosos accedan a activos de TI que, de otro modo, serían seguros.
- La **protección** de las **API** implica un conjunto de **procesos, prácticas y tecnologías** que defienden las interfaces de programación de aplicaciones (**API**) de los ataques y el uso indebido por parte de agentes maliciosos. La protección de las **API** es una parte esencial de un programa de **ciberseguridad, interfaces de programación de aplicaciones**, así como una prioridad máxima para los equipos de seguridad.
- Por lo tanto, **la protección de las API es fundamental** para mantener la seguridad de las redes y las aplicaciones, así como para evitar la exposición de los datos y otros problemas de seguridad.

6. ¿Qué es Postman???

Postman es una herramienta o aplicación gratuita que permite a los desarrolladores **interactuar con APIs externas, enviar peticiones y ver las respuestas de forma sencilla y eficiente**. No es específica de ningún lenguaje de programación o plataforma: tanto si construimos **APIs** a través de aplicaciones como ***React, Flask, Rails,*** etc. podemos utilizar **Postman** igualmente, ya que lo que **Postman** maneja son **peticiones de datos**. Por lo tanto es muy útil para construir **APIs**.

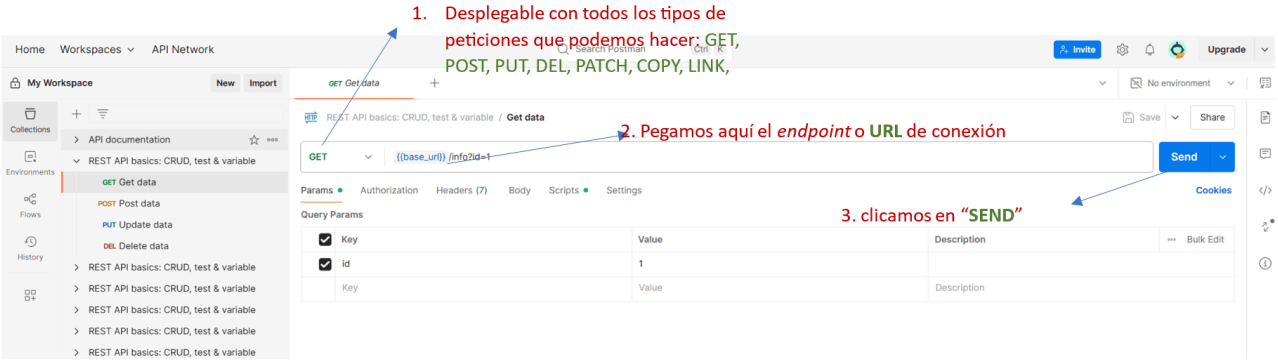
Postman es una plataforma integral para el **desarrollo y prueba** de **APIs**, que facilita cada etapa del ciclo de vida de las **API**, desde el **diseño** y las **pruebas** hasta la **entrega y la monitorización**.

Postman ofrece una interfaz gráfica de usuario intuitiva y fácil de usar, que permite a los desarrolladores interactuar con las **API** sin necesidad de una programación compleja.

Como comentábamos en el punto anterior, los datos que se obtienen directamente de la **API** (verbo: **GET**) pueden ser complejos de analizar:

Sin embargo, podemos optimizar la comunicación con la **API** y así poder obtener estos datos y traerlos a la pantalla de una forma que sea **más fácil de entender y manejar**:

Lo que haremos es poner la **URL** de la imagen anterior en la aplicación de **Postman**. Después de elegir el botón ***Request***, añadimos una descripción a esta petición, y entraríamos a la siguiente pantalla, que es donde nos comunicaremos con nuestra **API**:



4. Visualización de los datos obtenidos:

Como vemos en los datos que hemos obtenido a través de la herramienta de **Postman**, ahora los datos tienen otro aspecto: visualmente más analizables. A través de verbos ***HTTP***, como **GET, POST, PUT, DELETE.**, etc. esto ayuda a los desarrolladores a interactuar con las **APIs** y obtener resultados de este tipo.

Pruebas de API: Se utiliza para pruebas manuales, automatizadas y de carga, permitiendo validar la funcionalidad de las **API**, identificar errores y asegurar su correcto funcionamiento.

Herramienta versátil: **Postman** se integra con otras herramientas populares del desarrollo de software, como sistemas de control de versiones (**GitHub**), servicios de generación de documentación (**Swagger**) y herramientas de automatización de pruebas.

Con **Postman** se puede modificar cosas sobre la marcha: **podemos poner las URLs que deseemos modificar y ¡listo!**: mucho más sencillo que hacerlo en la URL misma o en la terminal. Pero también si construyes una API para alguien (por ejemplo utilizando **Flask API**), puedes hacer **pruebas** con ella **en tu servidor** (ya que se está ejecutando en tu **máquina local**) sin necesidad de implementarla cada vez que haces

pruebas. En realidad **te puedes comunicar con cualquier *endpoint*** con el que estés trabajando en tu máquina local.

---

7. ¿Qué es el polimorfismo?

Los términos de **herencia** y **polimorfismo** tiene sentido explicarlos de forma conjunta, ya que están conectados. Expliquemos primero el término de **herencia**.

7.1. Herencia

La herencia** ** es un proceso mediante el cual se puede crear una ***clase hija*** que hereda de una ***clase madre***, compartiendo sus **métodos y atributos**. Una ***clase hija*** **puede sobreescribir los métodos o atributos, o incluso definir unos nuevos**.

Se puede crear una ***clase hija*** con tan solo pasar como parámetro la clase de la que queremos heredar (***clase madre***). En el siguiente ejemplo, la clase hija ***Perro*** hereda de la clase madre ***Animal*** al incluirla, como decimos, en el parámetro de ***Perro***.

```
class Animal:
  """Clase base para mostrar la herencia"""

  def __init__(self, nombre, patas):
    self.nombre = nombre
    self.patas = patas

  def saluda(self):
    print("El animal llamado " + str(self.nombre) + " saluda")

class Perro(Animal):
  """Clase hija para mostrar la herencia"""

  def __init__(self, nombre):
    Animal.__init__(self, nombre, 4)
    self.sonido = "Guau"

  def ladra(self):
    print(self.sonido)

  def get_patas(self):
```

```
    return self.patas
```

```
mi_mascota = Perro("Chucho")
mi_mascota.saluda()
mi_mascota.ladra()
```

La instancia de la clase ***Perro*** :

```
mi_mascota = Perro("Chucho")
```

coge los atributos de ***nombre***, ***patas*** y el método ***saluda*** que tiene ***Animal***. Es como si internamente se copiara y pegara.

Si añadiéramos:

```
print(f"Mi mascota tiene {mi_mascota.get_patas()} patas.")
```

estaríamos llamando al método ***get_patas***, que devolvería el valor de ***patas*** y lo mostraría en pantalla.

7.2. Poliformismo

En Python, el **polimorfismo** es la capacidad de diferentes objetos de **responder de manera distinta a un mismo mensaje o método**. Esto significa que se puede **llamar al mismo método en diferentes clases** y que cada una de ellas ejecutará la lógica que le corresponda.

7.2.1. Poliformismo CON herencia

Ejemplo:

(En Python, la instrucción ***pass*** es una instrucción **nula** que no hace nada cuando se ejecuta. Se utiliza como un marcador de posición, especialmente cuando se requiere una sentencia sintácticamente pero no se necesita código real.)

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

    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        return "guau"
```

```
class Gato(Animal):
    def hacer_sonido(self):
        return "miau"

class Vaca(Animal):
    def hacer_sonido(self):
        return "mu"
```

Como vemos, tanto las clases (hijas) ***Perro***, ***Gato*** y ***Vaca*** llaman a la clase madre ***Animal*** y ejecutan sus particulares instrucciones dentro de cada una de ellas. Precisamente éste es el **polimorfismo** , cuya etimología (griega) se desglosaría en la palabra española "muchas" (***poli***) y la palabra "formas" (***morphe***), como resultado de llamar a la ***misma clase madre***. Lo que hemos hecho es **sobreescribir** el comportamiento de la ***clase madre*** con los diferentes sonidos "***guau", "miau", "mu"*** de las respectivas ***clases hijas***.

A veces no quieres que alguien llame a una ***clase en concreto*** (y esto es una convención común cuando utilizamos sistemas complejos o aplicaciones grandes). Por ejemplo, muchas veces no quieres que el usuario final se conecte ***con esa clase***. Por ello, creamos una clase **abstracta** con el único propósito de *almacenar el comportamiento compartido*, y solo las clases ***heredadas*** (las clases hijas) van a ser las que llamen ***a esta clase***. La forma de asegurarnos de que nadie llame ***a esta clase*** (ya que solo quiero que ***esa clase*** se pueda **heredar**) es declarando la función ***render()*** y la línea de comando ***raise NotImplementedError***. Esto protege a las clases que necesitemos proteger. Cada nueva clase que creemos, debe acceder primero a la función ***render()*** (y que contiene este " **NotImplementedError**"), y si no lo hiciéramos, nos un mensaje de ***error***, y nos diría que para que funcione el programa debemos primero acceder a esta función de ***render()***.

Ejemplos:

Comparemos este caso **A**:

```
class RectangularRoom(object):
    def __init__(self, width, height):
        raise NotImplementedError

    def cleanTileAtPosition(self, pos):
        raise NotImplementedError

    def isTileCleaned(self, m, n):
        raise NotImplementedError
```

Con este otro **B**:

```
class RectangularRoom(object):
    def __init__(self, width, height):
         pass

    def cleanTileAtPosition(self, pos):
         pass

    def isTileCleaned(self, m, n):
         pass
```

1. En el caso **B**: Si creas una subclase y olvidas indicarlo cómo hacerlo, ***isTileCleaned()***, o quizá no lo olvidas, pero lo escribes mal: ***isTileCLeaned()***, y entonces obtendrás **None** al llamarlo.
2. En el caso **A**: sin embargo, en este caso, ***raise NotImplementedError*** te obliga a implementarlo, ya que lanzará una excepción al intentar ejecutarlo hasta que lo hagas. Esto elimina muchos errores silenciosos (que por otra parte es muy típico en las personas).

7.2.2. Poliformismo SIN herencia

1. **No** hay clase **abstracta**.
2. Creamos **clases independientes**, con un método común, pero con implementaciones diferentes de ese método.
3. **Creamos una función** que es donde realmente comienza el **polimorfismo**.
4. **Instanciamos cada clase** con variables para cada clase independiente.
5. **Llamamos a la función**, cada vez con el **argumentos distintos**: con las variables que llamaban a cada una de las **clases independientes**:

Lo vemos en este ejemplo:

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

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

def hacer_ruido(animal):
    """
    1. Esta función acepta cualquier objeto que tenga un método 'hablar'
    y lo llama.  Esto demuestra polimorfismo sin herencia.
    """
    print(animal.hablar())

# 4. Creamos instancias de Perro y Gato
perro = Perro()
```

```
gato = Gato()

# 5. Llamamos a la función 'hacer_ruido' con ambos objetos
hacer_ruido(perro)  # Devuelve: Guau!
hacer_ruido(gato)  # Devuelve: Miau!
```

Qué tipo de polimormismo debemos utilizar (¿con o sin herencia?):

La buena práctica :

Si tenemos **bastante comportamiento compartido**, por ejemplo digamos que la clase principal (**Animal** en nuestro caso) tiene unas cuantas funciones dentro de ella, y ese comportamiento lo necesitan clases heredadas como *Perro* o *Gato* sería un caso donde nos vendría bien utilizar la **herencia**.

Si por el contrario no tenemos mucho comportamiento compartido y **queremos simplemente llamar a la misma función** (tal y como hemos visto en nuestro último ejemplo), y **no** es necesaria una **jerarquía de clases**, entonces utilizaríamos la opción de **polimorfismo con el enfoque de función**. Conseguimos tratar los objetos de una forma muy similar. A nuestra función ***hacer_ruido()*** no le importa qué tipo de clase estamos manejando. Si tenemos 20 tipos de clases que comparten la función ***hablar()***, es todo lo que importa.

7.2.3. La función *render()*

En el contexto de la programación orientada a objetos, la función de ***render()*** o representación visual, a menudo se implementa **de forma polimórfica**, permitiendo que diferentes tipos de objetos puedan ser dibujados o representados de manera diferente. Una vez más, esto significa que el método ***render()*** puede tener **diferentes comportamientos** dependiendo de la clase del objeto al que se aplica: **aunque la función se llame de la misma manera**, el código que se ejecuta dentro de ella variará, como hemos visto, según el tipo de objeto.

La función ***render()*** se utiliza comúnmente en el contexto de frameworks web para **renderizar vistas (templates)**, y puede ser aplicada polimóricamente en diferentes clases que representan diferentes tipos de vistas.

Ejemplo:

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

    def render(self):
        print("Dibujando figura (implementación general)")

class Rectangulo(FiguraGeometrica):
    def __init__(self, ancho, altura, nombre="Rectangulo"):
        super().__init__(nombre)
        self.ancho = ancho
        self.altura = altura

    def render(self):
        print(f"Dibujando rectángulo de {self.ancho} x {self.altura}")

class Circulo(FiguraGeometrica):
    def __init__(self, radio, nombre="Circulo"):
        super().__init__(nombre)
        self.radio = radio

    def render(self):
        print(f"Dibujando círculo de radio {self.radio}")

# Uso de polimorfismo
rectangulo = Rectangulo(5, 10)
circulo = Circulo(3)
figuras = [rectangulo, circulo]

for figura in figuras:
    figura.render()  # El método render() correcto será llamado para cada objeto
```

Devuelve:

```
Dibujando rectángulo de 5 x 10
Dibujando círculo de radio 3
```

En este ejemplo, ***render()*** es polimórfico porque las clases ***Rectangulo*** y ***Circulo*** implementan el método con **comportamientos diferentes**, que se activan según el objeto específico al que se llama. El polimorfismo, como hemos mencionado, permite que el código sea más flexible y fácil de mantener porque puedes usar objetos de diferentes clases de manera intercambiable sin tener que preocuparte por la implementación específica de cada uno.

---

8. ¿Que es un método Dunder?

En Python, este método se llama **Dunder** (o ***magic method***) es un método especial que se define con doble guion bajo (__) al principio y al final de su nombre. Estos métodos tienen un propósito específico, como **inicializar objetos**, **definir cómo se comportan con operadores**, o **dar una representación textual de un objeto**. Son llamados **automáticamente** por el intérprete de Python en ciertas situaciones.

Sirven para **personalizar el comportamiento de las clases y objetos**, y permiten que las clases se integren con las operaciones incorporadas de Python, como la suma, la comparación, o la impresión.

Tienen que ver con la forma en la que Python trabaja con métodos PRIVADOS y PROTEGIDOS dentro de sus CLASES. Algunos métodos **dunder** son:

Algunos métodos DUNDER:

- **__init__** : Ya hemos visto el **constructor** de la clase, que se ejecuta cuando se crea una instancia de la clase. Su principal función es **inicializar los atributos (los argumentos)** de esa instancia, es decir, darle un valor inicial a las variables que pertenecen a esa clase.
- **__str__** : Devuelve una **representación legible** de la clase, normalmente utilizada para **print()** ("legible para humanos en lugar de legible para máquinas"). Este método **retorna la representación de texto (cadena de caracteres) del objeto**, y es llamado cuando las funciones ***str()*** *-convertir un objeto en texto-* o ***print()*** *-muestra un objeto-* son llamadas en un objeto. Cuando llamamos ***str(mi_objeto)*** o ***print(mi_objeto)***, Python comprueba si la clase de ese objeto tiene un método **__str__** definido. Y recurre a **__repr__** para su representación.

```
##### Ejemplo:

Imaginemos que llamamos a una API para traernos datos (caso común); los coloco dentro de la clase, y SI NO ACTUALIZA en la base de datos, es que hay un problema o error y lo que hago es llamar a mi __str__
#mediante str(), y poder examinar todos los datos que provienen de la API,  y así ayuda a descubrir dónde está el error. (puedo no estar recibiendo los datos mal, o lo estoy analizando mal, )

    class Invoice:
      def __str__(self):
        return "This is the invoice class!"
```

```
inv = Invoice()     #instanciando.
print(str(inv))     #str() busca la definición del método str
                    #concretamente, lo que devuelve (return)
                    #(ayuda a tener visibilidad a lo que sea
                    # que hayas instanciado (en línea anterior)
```

Nota: *str()* Te proporciona la información sobre lo que la clase retorna. Normalmente se utiliza para propósitos de DEBUGGING, para que me muestre todo lo que contiene esa CLASE y mirar qué errores tiene (muestra atributos de datos).

- **__repr__** : Devuelve una representación legible o técnica de la clase, normalmente utilizada para **depuración** o **debugging**. Es decir se utiliza más para **raw output** (formato sin procesar), en su forma más pura, tal como la captura de una fuente sin caracteres especiales. P. ej. una salida a tus registros o a un registro de errores (error log). Si la clase tiene dentro pocos datos, no haría falta un **__repr__** , pero si tiene muchos datos, entonces sí. En resumen: cuando se usa **__repr__** : lo llamamos a través de la función ***repr()***, envolvemos todos los atributos, cualquier tipo de dato vaya a necesitar para hacer el **DEBUGGING**, ya que es una herramienta muy útil para los registros, pero también cuando intento arreglar un error en el programa. Ejemplo:

```
class Invoice:
  def __init__(self, client, total):
    self.client = client
    self.total = total

  def __str__(self):
    return f"Invoice from {self.client} for {self.total}"

  def __repr__(self):
    return f"Invoice({self.client}, {self.total})"
```

```
inv = Invoice('Google', 500)
print(str(inv))
print(repr(inv))
```

```
Nota:  Si no se define un método <font size= 4><span style = "color:blue"><strong> \_\_str__ 
</strong></span></font>() para una clase, entonces la implementación del objeto incorporado 
llama al método <font size= 4><span style = "color:blue"><strong> \_\_repr__ </strong> 
</span></font>() en su lugar.

Nosta2: En general, el método the <font size= 4><span style = "color:blue"><strong> 
\_\_str__ </strong></span></font>()  va dirigido a USUARIOS, y el método <font size= 4><span 
style = "color:blue"><strong> \_\_repr__ </strong></span></font>() a DESARROLLADORES.
```

- **__iter__** : retorna el objeto iterador (el que itera). El subsiguiente bucle subsiguiente llamará iterativamente al método **__next__** del objeto iterador para obtener valores. Por razones prácticas, la clase iterable puede implementar **__iter__** () y **__next__** () en la misma clase, y que **__iter__** () devuelva self.

```
##### Ejemplo:

    class MyIterable:
        def __init__(self, data):
            self.data = data
            self.index = 0

        def __iter__(self):
            return self

        def __next__(self):
            if self.index < len(self.data):
                value = self.data[self.index]
                self.index += 1
                return value
            else:
                raise StopIteration

    # Uso del iterador
    mi_iterable = MyIterable([1, 2, 3, 4, 5])

    # Iteración manual con next
    print(next(mi_iterable))  # Imprime 1
    print(next(mi_iterable))  # Imprime 2

    # Iteración con bucle for
    for item in mi_iterable:
        print(item)  # Imprime el resto de los elementos (3, 4, 5)
```

9. ¿Que es un decorador Python?

Antes de comenzar con los decoradores en sí, y para poder explicar los ejemplos de decoradores, haremos mención a algunos conceptos básicos.

Cuando hacemos codificación no solo programamos, también tiene importancia la FORMA en la que escribes, que debe COMUNICAR LA HISTORIA de ese programa: **como debería ser ejecutado, y como otros deberían utilizar ese código**. La CONVENCIONES COMUNES son una parte crucial.

En cuanto a los tipos de atributos en relación a su grado de protección, podemos encontrar información muy útil y explicativa en esta página web: https://medium.com/@jorge.cespedes.tapia/encapsulamiento-en-la-programaci%C3%B3n-orientada-a-objetos-poo-en-python-3ba757a93d17

- **UN GUION BAJO (_) : ATRIBUTO PROTEGIDO**

  Las **clases tienen acceso a clases superiores** (*p.ej. self.atributo1 = atributo1; self.atributo2 = atributo2*). Los datos, pues son accesibles DENTRO DE LA CLASE y SUBCLASES. Para permitir esto, debemos poner un guion bajo (_) delante del elemento o argumento a proteger (Variable de uso interno)

  Este guion bajo no es ninguna sintaxis especial, es una CONVENCIÓN que facilita a futuros desarrolladores o al creador/a del programa para mostrar que este atributo debe protegerse. Los GUIONES BAJOS en Python son una herramienta para la legibilidad y la comunicación entre programadores, más que para una restricción de acceso rígida.

- **DOS GUIONES BAJOS (__) : ATRIBUTO PRIVADO**

  Si tenemos un atributo (de datos) que solo debería ser accedido dentro solo de la clase, SUBCLASES NO ACCEDEN, esto se llama **ATRIBUTO PRIVADO**, y la CONVENCIÓN común para este caso es utilizar DOS GUONES BAJOS (__) delante del nombre de la variable. Python "envía" el nombre del atributo, modificándolo internamente para que sea más difícil acceder a él desde fuera de la clase.

En resumen:

PREFIJO	SIGNIFICADO	ACCESO FUERA DE LA CLASE
_	Sugiere uso interno	Posible, pero no recomendado
__	Indica privacidad. Transformación de nombres	Difícil (requiere conocer el nombre transformado). Técnicamente sí sería posible.

DECORADORES EN PYTHON

Antes de comenzar, tengamos en cuenta que **las funciones son ciudadanos de primera clase**, eso quiere decir que una función puede ser **asignada a una variable**, puede ser **utilizada como argumento** para otra función, o inclusive puede ser **retornada**.

Básicamente, un **decorador** no es más que una función la cual **toma como input una función** y a su vez **retorna otra función**. Lo que hacen en realidad es **modificar el comportamiento de otras funciones** y ayudan a **acortar nuestro código**. Esto es muy útil, principalmente, cuando queremos extender nuevas funcionalidades a dicha función. De allí el nombre **decorar**.

Ejemplo:

```
def poner_mayusculas(func):
    def envolver():
        return func().upper()

    return envolver

def saludo():
    return 'hola, estoy aprendiendo Python!'

saludo = poner_mayusculas(saludo)  #Esto es lo que decora (modifica) la función.

print(saludo())          # Devuelve:  HOLA, ESTOY APRENDIENDO PYTHON!
```

Pero haciéndolo de una forma más **escalable, elegante y profesional** lo haríamos de esta otra forma:

```
def poner_mayusculas(func):
    def envolver():
        return func().upper()

    return envolver

@poner_mayusculas
def saludo():
    return 'hola, estoy aprendiendo Python!'

print(saludo())           # Devuelve:  HOLA, ESTOY APRENDIENDO PYTHON!
```

Sintaxis: para decorar una función basta con colocar, en su parte superior de dicha función, el decorador con **el prefijo @**.

¿Se puede decorar varias veces una función en Python?

Sí podemos decorar varias veces una sola función. No obstante, esto ocurre **en el orden en que se llama a los decoradores**. Veamos un ejemplo en el que, además del decorador anterior, aplicamos uno que invierta el orden de las letras que le pasamos.

Ejemplo:

```
def poner_mayusculas(func):
    def envolver():
        return func().upper()

    return envolver

def invertir_str(func):
    def envolver():
        return func()[::-1]

    return envolver

@poner_mayusculas
@invertir_str
def saludo():
    return 'hola, estoy aprendiendo Python!'

print(saludo())            #Devuelve :  !NOHTYP ODNEIDNERPA YOTSE ,ALOH
```

Internamente, primero se aplica ***@poner_mayusculas*** para convertir todo el string en mayúsculas y luego ***@invertir_str*** para invertir el orden de la misma.

El decorador @property:

- Permite transformar un MÉTODO de clase en una PROPIEDAD de la clase, facilitando el acceso a su valor (a modo de *getter* y *setter*) como si fuera un atributo.Así el código es más limpio y legible. De esta forma, el código es máslimpio y legible. Se puede definir tres métodos para una propiedad:

Getter	Setter	Deleter
Captador: para acceder al valor del atributo	Establecedor: para fijar el valor del atributo	Eliminador: para eliminar el atributo de la instancia

Ejemplo:

```
class Casa:
    def __init__(self, precio):
        self._precio = precio

    @property
    def precio(self):
        return self._precio

    @precio.setter
    def precio(self, nuevo_precio):
        if nuevo_precio > 0 and isinstance(nuevo_precio, float):
            self._precio = nuevo_precio
        else:
            print("Introduzca un precio válido")

    @precio.deleter
    def precio(self):
        del self._precio
```

Nota: El precio está ahora **protegido** (tiene un guion bajo antes su nombre ***self._precio***) . El guion bajo indica a otros desarrolladores que **no se debe acceder a él ni modificarlo directamente fuera de la clase**. Solo se debe acceder a él mediante intermediarios (***getters*** y **setters**) si están disponibles.

- **Getter** o captador:
  - ***@property***- Se utiliza para indicar que vamos a **definir una propiedad**. El código se ve más legible y claro, ya que podemos ver claramente el propósito de este método.
  - ***def precio(self)*** El encabezado. El **getter** se llama exactamente igual que la propiedad que estamos definiendo: ***precio*** . Este es el nombre que usaremos para acceder y modificar el atributo fuera de la clase. El método solo acepta un parámetro formal, ***self***, que es una referencia a la instancia.
  - ***return self._precio*** Esta línea es exactamente lo que se esperaría de un **getter** normal. Se devuelve el valor del atributo **protegido**.
- **Setter** o establecedor:
  - ***@precio.setter***- Se usa para indicar que este es el método de establecimiento de la *propiedad* ***precio*** . No estamos usando *@property.setter* , sino ***@precio.setter*** . El nombre de la propiedad se incluye antes de ***.setter*** .
  - ***def precio(self, nuevo_precio)***: El encabezado y la lista de parámetros. El nombre de la propiedad se usa como nombre del **setter**. También tenemos un segundo parámetro formal ***(nuevo_precio)***, que es el nuevo valor que se asignará al atributo ***precio*** (si es válido).
  - Finalmente, tenemos el cuerpo del ***setter***, donde validamos el argumento para **comprobar si es un valor de punto flotante positivo**.
    - Si el argumento es válido, actualizamos el valor del atributo.
    - Si el valor no es válido, se imprime un mensaje descriptivo.

        - Nota: Se puede elegir cómo gestionar los valores no válidos según las necesidades de su programa.
- **Deleter** o eliminador:
    - ***@precio.deleter***- Se usa para indicar que este es el método de **eliminación de la propiedad precio**. Esta línea es muy similar a ***@precio.setter***, pero ahora estamos definiendo el método de eliminación, por lo que escribimos ***@precio.deleter*** .

Mientras que ***@property*** te permite definir atributos de SOLO LECTURA, ***@setter*** te permite definir la lógica de escritura o MODIFICACIÓN de atributos.

Cuando usas ***@setter***, defines un método que **se ejecuta automáticamente cuando intentas establecer el valor de un atributo en tu clase**. Se puede personalizar la lógica de **cómo se establece el valor del atributo**, lo que permite tener un mayor **control** sobre cómo se modifican los **atributos de nuestra clase**. Además, se puede combinar con ***@setter*** con ***@property*** para tener **aún más control** sobre la lógica de acceso y modificación de atributos en nuestra clase.

fin de los ejercicios teóricos del CheckPoint6