¿Para qué usamos Clases en Python?

Las clases en Python se utilizan para organizar y estructurar nuestro código de manera más eficiente. Una clase es como un plano o un molde que define cómo crear un objeto en Python.

Las clases nos permiten agrupar datos (atributos) y funciones (métodos) relacionadas en un solo lugar, lo que facilita la reutilización del código y la mantenibilidad del mismo. Son especialmente útiles cuando necesitamos crear múltiples objetos con las mismas características y comportamientos.

Para crear una clase, use la palabra clave class:

Ejemplo
Cree una clase llamada MiClase, con una propiedad llamada x:

```
class MiClase:
  x = 5
```

La sintaxis para definir una clase en Python es la siguiente:

```
class MiClase:
    def __init__(self, parametro1, parametro2):
        self.parametro1 = parametro1
        self.parametro2 = parametro2

    def mi_metodo(self):
        return self.parametro1 + self.parametro2
```

En este ejemplo, se define una clase llamada MiClase con un método constructor '__init__' que inicializa los atributos 'parametro1' y 'parametro2'. También se define un método 'mi_metodo' que simplemente devuelve la suma de los dos parámetros.

Ahora podemos usar la clase llamada MiClase para crear objetos.

Para utilizar esta clase y crear un objeto se haría de la siguiente manera:

```
objeto = MiClase(10, 20)
resultado = objeto.mi_metodo()
print(resultado)  # Output: 30
```

En este caso, se crea un objeto de la clase MiClase con los parámetros 10 y 20, y se llama al método 'mi_metodo' para obtener el resultado de la suma.

Métodos de objetos

Los objetos también pueden contener métodos. Los métodos en los objetos son funciones que pertenecen al objeto.
Creemos un método en la clase Persona:

Ejemplo
Inserte una función que imprima un saludo y ejecútela en el objeto p1:

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Nota: El selfparámetro es una referencia a la instancia actual de la clase y se utiliza para acceder a variables que pertenecen a la clase.

Modificar propiedades de objeto
Puede modificar propiedades en objetos como este:

Ejemplo
Establezca la edad de p1 en 40:

```
p1.age = 40
```

Eliminar propiedades del objeto
Puede eliminar propiedades de objetos utilizando la delpalabra clave:

Ejemplo
Elimine la propiedad de edad del objeto p1:

```
del p1.age
```

Eliminar objetos
Puede eliminar objetos utilizando la delpalabra clave:

Ejemplo
Eliminar el objeto p1:

```
del p1
```

La declaración del pase
Las definiciones no pueden estar vacías, pero si por alguna razón tiene una class definición sin contenido, introdúzcala en la pass declaración para evitar recibir un error.

Ejemplo

```
class Person:
  pass
```

En resumen, las clases en Python se utilizan para encapsular datos y funcionalidades relacionadas, permitiendo una mejor organización y estructuración del código, así como la creación de objetos con comportamientos específicos. Son una herramienta fundamental en la programación orientada a objetos en Python.

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

Cuando se crea una instancia de una clase en Python, automáticamente se ejecuta el **método '__init__'**. Este método, también conocido como el método de inicialización, se utiliza para inicializar los atributos de la clase y realizar cualquier otra configuración necesaria al momento de crear un objeto a partir de la clase.

El método ‘__init__’ se define dentro de la clase con una sintaxis específica, utilizando el nombre ‘__init__’ y con al menos un parámetro, generalmente llamado ‘self’, que hace referencia al objeto actual que está siendo creado. A través de este método, se pueden proporcionar argumentos adicionales para inicializar los atributos de la clase.

A continuación, se presenta un ejemplo de cómo se utiliza y se implementa el método ‘__init__’ en una clase en Python:

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

    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")

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

# Llamar al método saludar de la instancia persona1
persona1.saludar()
```

En este ejemplo, la clase ‘Persona’ tiene un método ‘__init__’ que recibe dos parámetros (‘nombre’ y ‘edad’) para inicializar los atributos ‘nombre’ y ‘edad’ de la clase. Al crear una instancia de la clase ‘Persona’ con los valores "Juan" y 30, se ejecuta automáticamente el método ‘__init__’ para inicializar los atributos de la instancia. Luego, se llama al método ‘saludar’ para mostrar un mensaje con el nombre y la edad de la persona.

Nota: La __init__() función se llama automáticamente cada vez que se utiliza la clase para crear un nuevo objeto.

La función __str__()

La función __str__() controla lo que se debe devolver cuando el objeto de clase se representa como una cadena.

Si la función __str__() no está configurada, se devuelve la representación de cadena del objeto:

Ejemplo

La representación de cadena de un objeto **SIN** la función __str__():

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

p1 = Person("John", 36)

print(p1)
```

Ejemplo

La representación de cadena de un objeto **CON** la función __str__():

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)
```

El autoparámetro

El self parámetro es una referencia a la instancia actual de la clase y se utiliza para acceder a variables que pertenecen a la clase.

No es necesario que tenga nombre self, puedes llamarlo como quieras, pero tiene que ser el primer parámetro de cualquier función de la clase:

Ejemplo

Utilice las palabras *mysillyobject* y *abc* en lugar de *self* :

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

El método '__init__' se ejecuta automáticamente al crear una instancia de una clase en Python, se utiliza para inicializar los atributos de la clase y se define con una sintaxis específica dentro de la clase. Su principal objetivo es configurar el estado inicial de los objetos creados a partir de la clase.

¿Cuáles son los tres verbos de API?

En el contexto de programación y desarrollo de aplicaciones, los tres verbos API más comunes son GET, POST y PUT. Estos verbos son utilizados para realizar diferentes acciones sobre recursos en un servidor o base de datos a través de una API (Interfaz de Programación de Aplicaciones).

1. **GET**: Este verbo se utiliza para obtener información de un recurso específico en la API. Por ejemplo, si queremos obtener los datos de un usuario con ID 1, podríamos hacer una solicitud GET a la URL "/usuarios/1". Esta solicitud devolvería la información del usuario con ID 1 en formato JSON o XML, dependiendo de la configuración de la API.

Para utilizar el verbo GET, la sintaxis es la siguiente:

```
GET /usuarios/1
```

Este verbo es ampliamente utilizado para consultar información en la API y no suele modificar los datos del servidor.

2. **POST**: El verbo POST se utiliza para crear un nuevo recurso en la API. Por ejemplo, si queremos agregar un nuevo usuario a la base de datos, podríamos hacer una solicitud POST a la URL "/usuarios" con los datos del nuevo usuario en el cuerpo de la solicitud. La API entonces procesaría la solicitud y crearía un nuevo usuario en la base de datos.

La sintaxis para utilizar el verbo POST es la siguiente:

```
POST /usuarios
{
 "nombre": "Juan",
 "edad": 30,
 "email": "juan@example.com"
}
```

Este verbo se utiliza para enviar datos al servidor y modificar la base de datos de la API.

3. **PUT**: El verbo PUT se utiliza para modificar un recurso existente en el servidor. Se utiliza cuando se desea actualizar la información de un usuario o editar un artículo en la API.

Ejemplo de solicitud PUT:

```
PUT /api/usuarios/1 HTTP/1.1
Host: api-ejemplo.com
Content-Type: application/json

{
 "nombre": "Juan Carlos",
 "apellido": "Perez",
 "email": "juan.carlos@example.com"
}
```

Los verbos de API (GET, POST, PUT) permiten realizar operaciones de consulta, creación y actualización de datos en una API RESTful, proporcionando una manera estándar de interactuar con los recursos del servidor. Estos verbos son fundamentales para el desarrollo y consumo de APIs, ya que permiten una comunicación eficiente y coherente entre clientes y servidores web. Es importante tener en cuenta que estos verbos deben utilizarse de acuerdo con las convenciones y estándares de la API para garantizar su correcto funcionamiento y seguridad.

¿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 tablas y filas para almacenar datos, MongoDB utiliza un enfoque basado en documentos JSON. Esto significa que los datos se almacenan de forma más flexible, lo que permite una mayor escalabilidad y rendimiento en entornos donde se manipulan grandes cantidades de información.

Una de las razones por las que se utiliza MongoDB es su **capacidad para manejar datos no estructurados o semiestructurados de manera eficiente**. Por ejemplo, en una aplicación web donde se almacenan datos de usuarios, cada usuario puede tener diferentes campos en su perfil (como nombre, edad, dirección, etc.). En una base de datos SQL, se necesitarían múltiples tablas relacionadas para almacenar esta información, mientras que en MongoDB todo se puede guardar en un solo documento.

Una de las características más destacadas de MongoDB es su **capacidad de escalabilidad**. Esta base de datos puede manejar grandes cantidades de datos distribuidos en múltiples servidores, lo que la hace ideal para aplicaciones que requieren un alto rendimiento y disponibilidad. MongoDB utiliza la técnica de fragmentación para distribuir los datos en diferentes nodos, lo que permite que la carga de trabajo se distribuya de manera equitativa y que se pueda escalar horizontalmente agregando más servidores según sea necesario.

Otra ventaja de MongoDB es su **soporte para la indexación de datos**. Los índices en MongoDB permiten acelerar las consultas al especificar los campos que deben ser indexados, lo que mejora el rendimiento de las operaciones de lectura. Además,

MongoDB ofrece un conjunto de herramientas integradas para la administración de la base de datos, como la posibilidad de realizar copias de seguridad, monitorear el rendimiento y realizar ajustes para optimizar el funcionamiento del sistema.

En cuanto a la seguridad, MongoDB **ofrece diversas funcionalidades para proteger los datos almacenados en la base de datos**. Entre estas funcionalidades se incluye la autenticación de usuarios, la autorización basada en roles y privilegios, el cifrado de datos en reposo y en tránsito, y la auditoría de acciones realizadas en la base de datos.

La sintaxis de MongoDB es similar a la de JavaScript, lo que facilita su aprendizaje para aquellos familiarizados con este lenguaje de programación.

A continuación, se describen algunas de **las funcionalidades más importantes de MongoDB**, junto con ejemplos y explicaciones detalladas.

1. Creación de bases de datos y colecciones

Una de las primeras tareas que se realizan en MongoDB es la creación de bases de datos y colecciones para organizar los datos de forma adecuada. La sintaxis para crear una nueva base de datos es la siguiente:

```
use mi_basededatos
```

Una vez creada la base de datos, se pueden crear colecciones dentro de ella utilizando el método ‘db.createCollection()’:

```
db.createCollection("mi_coleccion")
```

2. Inserción de documentos

Para insertar datos en una colección de MongoDB, se utiliza el método ‘db.collection.insertOne()’ o ‘db.collection.insertMany()’, dependiendo de si se quiere insertar un solo documento o varios documentos a la vez. A continuación se muestra un ejemplo de inserción de un documento en una colección:

```
db.mi_coleccion.insertOne({
  nombre: "Juan",
```

```
  edad: 30,
  ciudad: "Madrid"
})
```

3. Consultas

MongoDB permite realizar consultas avanzadas utilizando su potente sistema de consultas basado en documentos JSON. Para realizar consultas en MongoDB, se utiliza el método ‘db.collection.find()’:

```
db.mi_coleccion.find({ ciudad: "Madrid" })
```

Esta consulta buscará todos los documentos en la colección ‘mi_coleccion’ cuyo campo ‘ciudad’ tenga el valor "Madrid".

4. Actualización de documentos

Para actualizar documentos en MongoDB, se utiliza el método ‘db.collection.updateOne()’ o ‘db.collection.updateMany()’, dependiendo de si se quiere actualizar un solo documento o varios documentos a la vez. A continuación se muestra un ejemplo de actualización de un documento en una colección:

```
db.mi_coleccion.updateOne(
  { nombre: "Juan" },
  { $set: { edad: 31 } }
)
```

En este caso, se actualiza el documento cuyo campo ‘nombre’ sea "Juan", modificando su campo ‘edad’ a 31.

5. Eliminación de documentos

Para eliminar documentos en MongoDB, se utiliza el método ‘db.collection.deleteOne()’ o ‘db.collection.deleteMany()’, dependiendo de si se quiere eliminar un solo documento o varios documentos a la vez. A continuación se muestra un ejemplo de eliminación de un documento en una colección:

```
db.mi_coleccion.deleteOne({ nombre: "Juan" })
```

En este ejemplo, se elimina el documento cuyo campo ‘nombre’ sea "Juan" de la colección ‘mi_coleccion’.

MongoDB es una base de datos NoSQL que se utiliza para almacenar y manipular datos de forma flexible y eficiente. Es una excelente opción para aquellas aplicaciones que manejan grandes volúmenes de datos y que requieren una estructura de datos flexible y dinámica. Su sintaxis simple y su capacidad para manejar datos no estructurados lo convierten en una opción popular para aplicaciones modernas que requieren escalabilidad y rendimiento.

¿Qué es una API?

Una API, o Interfaz de Programación de Aplicaciones, es un conjunto de reglas y herramientas que permiten a diferentes programas comunicarse entre sí de manera eficiente. En otras palabras, una API actúa como un intermediario que permite que dos aplicaciones se conecten y compartan información de manera segura.

Las APIs se utilizan en una variedad de situaciones, desde redes sociales como Facebook y Twitter que permiten a los desarrolladores acceder a sus plataformas y crear aplicaciones conectadas, hasta servicios de mapas como Google Maps que brindan información geoespacial a otras aplicaciones.

Existen diferentes tipos de APIs en Python, como por ejemplo las **APIs de librerías**, que permiten utilizar funciones predefinidas en una librería; las **APIs web**, que permiten acceder y consumir información de un servidor remoto a través de internet; y las **APIs de servicios**, que proporcionan acceso a funcionalidades específicas de una aplicación.

La sintaxis de una API suele estar definida por el proveedor del servicio y puede variar dependiendo de la tecnología utilizada. Por ejemplo, la API de Twitter requiere que los desarrolladores autentifiquen sus solicitudes utilizando tokens de acceso, mientras que la API de Google Maps utiliza coordenadas geográficas para buscar

ubicaciones específicas. Pero en general se suelen usar funciones y métodos que reciben parámetros de entrada y devuelven resultados.

Cómo utilizar la API en Python

Para utilizar la API en Python, primero debes importar el módulo que contiene la API que quieres utilizar. Por ejemplo, para utilizar la API de la base de datos MySQL, debes importar el módulo mysql.connector.

Una vez que hayas importado el módulo, puedes utilizar las funciones y clases del módulo para acceder a los datos, realizar operaciones y controlar dispositivos.

Ejemplos de API en Python

Hay muchas API disponibles para Python. Algunos ejemplos de API populares incluyen:

- **API de la base de datos MySQL:** Esta API permite a los programadores acceder a datos almacenados en una base de datos MySQL.
- **API de la API de Google Maps:** Esta API permite a los programadores obtener información sobre mapas y direcciones.
- **API de la API de Twitter:** Esta API permite a los programadores acceder a datos de Twitter.

Guía de API en Python

A continuación, se incluye una guía paso a paso sobre cómo utilizar la API en Python:

1. Importa el módulo que contiene la API que quieres utilizar.
2. Crea una instancia del objeto de la API.
3. Llama a los métodos del objeto de la API para acceder a los datos, realizar operaciones y controlar dispositivos.

Ejemplo de código

El siguiente código muestra cómo utilizar la API de la base de datos MySQL para conectarse a una base de datos y recuperar los datos de una tabla:

```
import mysql.connector

# Conecta a la base de datos
connection = mysql.connector.connect(
  host="localhost",
```

```
    user="root",
    password="password",
    database="my_database"
)

# Crea un cursor para ejecutar consultas
cursor = connection.cursor()

# Ejecuta la consulta
cursor.execute("SELECT * FROM my_table")

# Obtiene los resultados de la consulta
results = cursor.fetchall()

# Imprime los resultados
for result in results:
    print(result)
```

Este código se conectará a la base de datos my_database en el host localhost. Luego, creará un cursor para ejecutar consultas. A continuación, ejecutará la consulta SELECT * FROM my_table para recuperar todos los datos de la tabla my_table. Finalmente, imprimirá los resultados de la consulta.

En resumen, una API es un conjunto de reglas y mecanismos que permiten a los programas comunicarse entre sí de manera eficiente. Se utiliza para facilitar la integración de diferentes sistemas y servicios, permitiendo a los desarrolladores acceder a funcionalidades específicas de una plataforma de manera segura y controlada.

¿Qué es Postman?

Postman es una herramienta de desarrollo de API que se utiliza para probar, desarrollar y documentar APIs de forma sencilla y eficiente. Su principal objetivo es permitir a los desarrolladores realizar diversas operaciones HTTP, como enviar solicitudes, validar respuestas y automatizar pruebas.

Postman es ampliamente utilizado en el desarrollo de API ya que brinda una interfaz gráfica intuitiva para interactuar con endpoints, lo que facilita la creación de solicitudes y pruebas de manera rápida y efectiva. Además, permite organizar y almacenar solicitudes en colecciones, lo que facilita la colaboración y el intercambio de información entre equipos de desarrollo.

La sintaxis en Postman es muy sencilla y se basa en el uso de solicitudes HTTP, como GET, POST, PUT y DELETE. Por ejemplo, para enviar una solicitud GET a una API, se debe especificar la URL del endpoint y posibles parámetros en la barra de dirección de Postman. Además, se pueden añadir encabezados, autenticaciones y body en el caso de solicitudes POST.

Postman es útil para realizar pruebas de integración, validación de respuestas, automatización de pruebas y generación de documentación de API. Además, ofrece funcionalidades avanzadas como monitoreo de endpoints, colaboración en equipo y simulación de entornos para probar diferentes escenarios.

En conclusión, Postman es una herramienta esencial para cualquier desarrollador que trabaje con APIs, ya que facilita el proceso de desarrollo y prueba de endpoints de forma eficiente y ordenada.

¿Qué es el polimorfismo?

La palabra "polimorfismo" significa "muchas formas" y en programación se refiere a métodos/funciones/operadores con el mismo nombre que se pueden ejecutar en muchos objetos o clases. El polimorfismo es un concepto de la programación orientada a objetos que permite que un objeto pueda comportarse de diferentes maneras dependiendo del contexto en el que se encuentre. En Python, el polimorfismo se refiere a la capacidad de un objeto para utilizar métodos de la clase padre o de la propia clase de manera dinámica, es decir, en tiempo de ejecución.

La sintaxis para implementar el polimorfismo en Python es sencilla. Basta con definir una clase base con métodos que puedan ser sobrescritos por sus subclases, y luego crear varias subclases que redefinan esos métodos según sea necesario.

Polimorfismo de función

Un ejemplo de una función de Python que se puede utilizar en diferentes objetos es la len()función.

Cadena

Para cadenas len()devuelve el número de caracteres:

Ejemplo

```
x = "Hello World!"

print(len(x))
```

Tupla

Para tuplas len()devuelve el número de elementos de la tupla:

Ejemplo

```
mytuple = ("apple", "banana", "cherry")

print(len(mytuple))
```

Diccionario

Para diccionarios, len()devuelve el número de pares clave/valor en el diccionario:

Ejemplo

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

print(len(thisdict))
```

Polimorfismo de clase

El polimorfismo se usa a menudo en métodos de clase, donde podemos tener varias clases con el mismo nombre de método.

Por ejemplo, digamos que tenemos tres clases: Car, Boaty Plane, y todas tienen un método llamado move():

Ejemplo

Diferentes clases con el mismo método:

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Drive!")

class Boat:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Sail!")

class Plane:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")       #Create a Car class
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class
plane1 = Plane("Boeing", "747")     #Create a Plane class

for x in (car1, boat1, plane1):
  x.move()
```

Mira el bucle for al final. Debido al polimorfismo podemos ejecutar el mismo método para las tres clases.

Polimorfismo de clase de herencia

¿Qué pasa con las clases con clases secundarias con el mismo nombre? ¿Podemos usar polimorfismo allí?
Sí. Si usamos el ejemplo anterior y creamos una clase principal llamada Vehicle y creamos Car clases Boat secundarias Plane de Vehicle, las clases secundarias heredan los Vehicle métodos, pero pueden anularlos:

Ejemplo
Cree una clase llamada Vehicle y cree Car clases Boat secundarias Plane de Vehicle:

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Move!")

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

Las clases secundarias heredan las propiedades y métodos de la clase principal.

En el ejemplo anterior puedes ver que la Car clase está vacía, pero hereda brand, model y move() de Vehicle. Las clases Boat y Plane también heredan brand, model y move() de Vehicle, pero ambas anulan el move() método. Debido al polimorfismo podemos ejecutar el mismo método para todas las clases.

En resumen, el polimorfismo en Python es una característica poderosa que permite que los objetos se comporten de manera diferente en función de su tipo concreto, lo que facilita la escritura de código modular, flexible y reutilizable.

¿Qué es un método dunder?

Un método dunder es un método especial de Python que comienza y termina con doble guion bajo (__). Estos métodos son utilizados para realizar operaciones especiales, como la comparación de objetos, la asignación de valores predeterminados, la conversión de objetos a cadenas, entre otros.

Un ejemplo de un método dunder es **'__init__'**, que se utiliza para inicializar un objeto cuando se crea una instancia de una clase. Por ejemplo:

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

persona1 = Persona("Juan", 30)
```

En este caso, el método '__init__' se ejecuta automáticamente cuando se crea una nueva instancia de la clase 'Persona', y se encarga de asignar los valores proporcionados a las variables de instancia 'nombre' y 'edad'.

Otro ejemplo de un método dunder es **'__eq__'**, que se utiliza para comparar objetos. Por ejemplo:

```
class Punto:
   def __init__(self, x, y):
```

```
        self.x = x
        self.y = y

    def __eq__(self, otro_punto):
        return self.x == otro_punto.x and self.y == otro_punto.y

punto1 = Punto(1, 2)
punto2 = Punto(1, 2)

print(punto1 == punto2) # True
```

En este caso, el método '__eq__' se encarga de comparar dos objetos de la clase 'Punto' para determinar si son iguales, en base a las coordenadas 'x' y 'y' de cada punto.

Otros métodos Dunder más populares son:

'__str__': Este método se utiliza para devolver una representación legible de un objeto como una cadena de texto. Es comúnmente utilizado para la impresión de un objeto. Su sintaxis es la siguiente:

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

    def __str__(self):
        return f"{self.nombre} tiene {self.edad} años"

persona = Persona("Juan", 25)
print(persona)  # Imprime: Juan tiene 25 años
```

'__repr__': Similar a '__str__', este método se utiliza para devolver una representación de un objeto como una cadena de texto, pero se utiliza más para debugging y desarrollo. La diferencia principal es que '__repr__' debe ser una representación que reproduzca el objeto si se pasa a 'eval()'. Su sintaxis es la siguiente:

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Punto({self.x}, {self.y})"

p = Punto(2, 3)
print(p)  # Imprime: Punto(2, 3)
```

‘__iter__’ y **‘__next__’**: Estos métodos se utilizan para hacer que un objeto sea iterable, es decir, puedas iterar sobre él usando un bucle ‘for’. ‘__iter__’ debe devolver un objeto iterador (por lo general el propio objeto), y ‘__next__’ debe devolver el siguiente elemento en la iteración. Ejemplo:

```
class NumerosPares:
    def __init__(self, max):
        self.max = max
        self.actual = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.actual < self.max:
            self.actual += 2
            return self.actual
        else:
            raise StopIteration

numeros_pares = NumerosPares(10)
for num in numeros_pares:
    print(num)  # Imprime los números pares del 0 al 10
```

Los métodos Dunder en Python son métodos especiales que permiten personalizar el comportamiento de una clase y proporcionan funcionalidades adicionales. Al implementar estos métodos en una clase, se puede hacer que los objetos de esa clase se comporten de manera más intuitiva y eficiente en diferentes situaciones.

¿Qué es un decorador de python?

Un decorador en Python es una función que permite modificar o extender el comportamiento de otra función sin modificar su código. Los decoradores son funciones de orden superior, es decir, funciones que toman como argumento otra función y devuelven otra función.

La principal característica de un decorador es que envuelve la función original y le agrega funcionalidad adicional. Esto permite reutilizar código, simplificar el mantenimiento y mejorar la legibilidad del código.

Un ejemplo sencillo de decorador en Python sería el siguiente:

```
def decorador(funcion):
    def nueva_funcion():
        print("Antes de ejecutar la función")
        funcion()
        print("Después de ejecutar la función")
    return nueva_funcion

@decorador
def mi_funcion():
    print("En la función original")

mi_funcion()
```

En este caso, al llamar a ‘mi_funcion’, se ejecuta la función envuelta por el decorador, lo que permite imprimir en consola un mensaje antes y después de ejecutar la función original.

Los decoradores son ampliamente utilizados en Python para realizar tareas como la validación de parámetros, el acceso a bases de datos, la gestión de logs, la medición de tiempo de ejecución, entre otros.

La sintaxis para aplicar un decorador en Python es utilizando el signo '**@**' seguido del nombre del decorador sobre la función que se desea decorar. Esto se conoce como decorador de función.

Además, también es posible crear decoradores de clase y decoradores parametrizados, lo que permite extender aún más su funcionalidad.

Un **decorador de clase** en Python es una función que recibe una clase como argumento y devuelve una clase nueva que extiende o modifica la funcionalidad de la clase original. Esto se logra mediante la definición de una nueva clase que hereda de la clase original y, en el proceso, se pueden agregar nuevos métodos, alterar atributos o sobrescribir métodos existentes.

Un ejemplo de un decorador de clase sería el siguiente:

```
def decorador_clase(cls):
    class NuevaClase(cls):
        def nuevo_metodo(self):
            print("Este es un nuevo método añadido mediante el decorador de clase")

    return NuevaClase

@decorador_clase
class MiClase:
    def __init__(self, nombre):
        self.nombre = nombre

objeto = MiClase("Ejemplo")
objeto.nuevo_metodo()  # Output: Este es un nuevo método añadido mediante
el decorador de clase
```

Por otro lado, los **decoradores parametrizados** son funciones que devuelven un decorador regular y permiten configurar o personalizar la decoración aplicada a una

función o clase. Estos decoradores toman argumentos adicionales que se utilizan para modificar su comportamiento.

Un ejemplo de un decorador parametrizado sería el siguiente:

```
def decorador_parametrizado(parametro):
    def decorador(func):
        def wrapper():
            print(parametro)
            func()
        return wrapper
    return decorador

@decorador_parametrizado("Este es un mensaje personalizado")
def funcion():
    print("Esta es una función decorada")

funcion()  # Output: Este es un mensaje personalizado \n Esta es una función decorada
```

En este ejemplo, el decorador ‘decorador_parametrizado’ toma un parámetro que se utilizará en la función envoltorio ‘wrapper’ para imprimir un mensaje personalizado antes de llamar a la función original.

En conclusión, un decorador en Python es una herramienta poderosa que permite modificar el comportamiento de una función sin modificar su código, lo que facilita la reutilización y mantenimiento del mismo. Su sintaxis sencilla y versatilidad lo convierten en una herramienta muy útil para programadores Python.