¿Para qué usamos Clases en Python?

En Python, las clases son utilizadas para organizar y estructurar el código de manera más efectiva. Permiten agrupar datos y funciones relacionadas en un solo lugar, facilitando la reutilización de código y la creación de programas más mantenibles. A continuación diferentes aplicaciones de las clases:

1. **Abstracción y Modelado:**
    - *Ejemplo:* Supongamos que estás creando un programa para gestionar una biblioteca. Puedes tener una clase **Libro** que represente las propiedades y acciones relacionadas con los libros, como el título, autor, y métodos para prestar y devolver.
2. **Encapsulación:**
    - *Ejemplo:* Una clase puede ocultar la complejidad interna y exponer solo la interfaz necesaria. Por ejemplo, una clase **Coche** podría tener métodos como **arrancar** y **detener**, ocultando la complejidad interna del motor.
3. **Herencia:**
    - *Ejemplo:* Si tienes una clase **Vehiculo**, puedes crear subclases como **Coche** y **Moto** que hereden las características generales de **Vehiculo** y luego agregar características específicas a cada tipo de vehículo.
4. **Polimorfismo:**
    - *Ejemplo:* Puedes usar una interfaz común para diferentes clases. Por ejemplo, una clase **Animal** podría tener un método **hacer_sonido**, y las subclases como **Perro** y **Gato** podrían implementar ese método de manera diferente.

En resumen, las clases en Python nos permiten organizar el código de manera más eficiente, proporcionando una estructura clara y facilitando la reutilización del código. Son fundamentales para la programación orientada a objetos, un paradigma ampliamente utilizado en el desarrollo de software.

Ejemplo de clase:

```
class Invoice:
  def __init__(self, client, total): #creamos la instancia para la clase con el constructor  __init__
    self.client = client
    self.total = total
  def formatter(self):
    return f'{self.client} owes: ${self.total}'
google = Invoice('Google', 100) # creamos con __init__las variables self
snapchat = Invoice('SnapChat', 200)
print(google.formatter()) # Google owes: $100 -> hemos llevado las variables self a la otra función
print(snapchat.formatter()) # SnapChat owes: $200
```

- Se define un constructor (**__init__**) que se ejecuta automáticamente cuando se crea una nueva instancia de la clase.
- El constructor toma dos parámetros (**client** y **total**) y asigna estos valores a los atributos (**self.client** y **self.total**) de la instancia.
- Se define un método llamado **formatter** que toma **self** (la instancia) como primer parámetro.

- Este método devuelve una cadena formateada que indica quién debe cuánto en la factura.
- Se crean dos instancias de la clase **Invoice** (facturas), una para Google y otra para Snapchat.
- Se utilizan los valores proporcionados para **client** y **total** en el constructor (**__init__**) al crear las instancias.
- Se llama al método **formatter** en cada instancia de la clase.
- Imprime la información formateada sobre cuánto debe cada cliente.

Para saber más:

https://docs.python.org/es/3/tutorial/classes.html

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

Cuando se crea una instancia de una clase en Python, el método que se ejecuta automáticamente es el método **__init__**, también conocido como el "constructor". ¿como funciona?

1. **Inicialización de Atributos:**
    - El método **__init__** se utiliza para inicializar los atributos de la instancia con valores específicos cuando se crea.
    - Permite configurar el estado inicial de los objetos.
2. **Ejecución Automática:**
    - Se ejecuta automáticamente cada vez que se crea una nueva instancia de la clase.
    - Proporciona la oportunidad de realizar acciones de configuración necesarias al crear objetos.

En el ejemplo de la cuestión anterior se puede ver el funcionamiento de una instancia de clase.

Para saber más:

https://docs.python.org/es/3/tutorial/classes.html

¿Cuáles son los tres verbos de API?

Los tres verbos principales en el contexto de las API (Interfaz de Programación de Aplicaciones) son:

Los tres verbos principales en el contexto de las API (Interfaz de Programación de Aplicaciones) son:

1. **GET (Obtener):**
    - **Uso:** Se utiliza para recuperar información o datos de un recurso específico.
    - **Ejemplo:** Obtener información de un usuario en una aplicación web.
2. **POST (Enviar/Crear):**
    - **Uso:** Se utiliza para enviar datos al servidor para crear un nuevo recurso.
    - **Ejemplo:** Crear un nuevo comentario en un blog.
3. **PUT/PATCH (Actualizar/Modificar):**
    - **Uso:** Se utilizan para actualizar o modificar un recurso existente en el servidor.
    - **Ejemplo:** Actualizar la información de un producto en un sistema de comercio electrónico.

¿Es MongoDB una base de datos SQL o NoSQL?

MongoDB es una base de datos NoSQL.

¿Qué significa NoSQL? "NoSQL" significa "Not Only SQL", lo que indica que no se adhiere a las estructuras y restricciones tradicionales de las bases de datos SQL (Relacionales). En lugar de utilizar tablas y esquemas predefinidos, las bases de datos NoSQL permiten almacenar y recuperar datos en formatos más flexibles.

¿Por qué MongoDB es NoSQL? MongoDB utiliza un modelo de datos NoSQL llamado "document-oriented", donde la información se organiza y almacena en documentos BSON (Binary JSON) dentro de colecciones. Este enfoque proporciona flexibilidad y escalabilidad, ya que no requiere un esquema fijo y puede manejar grandes cantidades de datos no estructurados.

¿Qué es una API?

Una API (Interfaz de Programación de Aplicaciones) es un conjunto de reglas y herramientas que permite a diferentes aplicaciones comunicarse entre sí. Es como un camarero en un restaurante: toma tus pedidos (solicitudes) y trae la comida de la cocina (respuestas). En el desarrollo de software, las APIs facilitan la interacción y el intercambio de información entre distintas aplicaciones.

¿Por qué se utiliza una API?

1. **Interconexión de Aplicaciones:** Permite que diferentes aplicaciones se comuniquen y compartan datos entre sí, incluso si están construidas con tecnologías diferentes.
2. **Reutilización de Funcionalidades:** Facilita la reutilización de funcionalidades de una aplicación en otra sin tener que compartir todo el código fuente.
3. **Desarrollo Rápido:** Al utilizar APIs, los desarrolladores pueden integrar rápidamente funciones existentes en lugar de crear todo desde cero.

¿Qué es Postman?

Postman es una herramienta que facilita el desarrollo y la prueba de APIs (Interfaz de Programación de Aplicaciones). Es como un caja de herramientas para los desarrolladores, proporcionando una interfaz gráfica amigable para enviar solicitudes HTTP a APIs, revisar las respuestas y colaborar en proyectos relacionados con APIs.

¿Por qué se utiliza Postman?

1. **Pruebas de APIs:** Permite a los desarrolladores probar sus APIs al enviar solicitudes y recibir respuestas de manera interactiva.
2. **Documentación:** Facilita la creación de documentación de APIs al permitir a los equipos compartir y entender las diferentes rutas, parámetros y respuestas.
3. **Colaboración:** Permite a los equipos colaborar en el desarrollo de APIs al compartir colecciones de solicitudes y entornos.

Para saber más:

https://www.postman.com/

¿Qué es el polimorfismo?

En Python, el polimorfismo es un concepto que permite a un objeto tomar múltiples formas. En otras palabras, un mismo nombre (método o función) puede tener diferentes comportamientos según el tipo de objeto al que se aplica.

¿Por qué se utiliza el Polimorfismo?

1. **Flexibilidad y Adaptabilidad:** Permite que un mismo método sea utilizado con diferentes tipos de objetos, proporcionando flexibilidad y adaptabilidad en el código.
2. **Facilita la Implementación de Interfaces Comunes:** Facilita la creación de interfaces comunes para distintas clases, lo que simplifica el diseño y la implementación de software.

Ejemplo Práctico: uso de poliformismo para realizar un generador HTML:

Este código ilustra cómo la herencia y el polimorfismo permiten que diferentes clases compartan un mismo nombre de método (**render**), pero cada clase puede tener su propia implementación específica. En este caso, las clases **Heading** y **Div** heredan de la clase base **Html** y proporcionan sus propias versiones del método **render**.

```
class Html:
    def __init__(self, content):
        self.content = content
    def render(self):
        raise NotImplementedError("Subclass must implement render method") # Aviso para no llamar a esta clase
```

```python
class Heading(Html): #entre paréntesis indicamos la función padre.
    def render(self): #sobreescribimos el comportamiento del padre
        return f'<h1>{self.content}</h1>'
class Div(Html):
    def render(self): #sobreescribimos el comportamiento del padre
        return f'<div>{self.content}</div>'

tags = [Div('Some content'), Heading('My Amazing Heading'), Div('Another div')]
for tag in tags:
    print(tag.render())  #<div>Some content</div>
                         #<h1>My Amazing Heading</h1>
                         #<div>Another div</div>
```

- **Html** es una clase base que contiene un constructor (**__init__**) para inicializar el contenido y un método **render** que genera HTML.
- El método **render** tiene un mensaje de error (**NotImplementedError**) para indicar que cualquier subclase debe implementar este método.
- **Heading** es una subclase de **Html** que hereda su constructor y método **render**.
- Sobreescribe el método **render** para generar un encabezado HTML (**<h1>**).
- **Div** es otra subclase de **Html** que hereda su constructor y método **render**.
- Sobreescribe el método **render** para generar una etiqueta de div HTML (** **).
- Se crean instancias de las clases **Div** y **Heading** con contenido específico.
- Se almacenan en la lista **tags**.
- Se itera sobre la lista y se llama al método **render** de cada objeto, mostrando el contenido HTML generado.

¿Qué es un método dunder?

Un método dunder en Python se refiere a un método especial que utiliza doble guion bajo (__) al principio y al final de su nombre. "Dunder" es una abreviatura de "double underscore" (doble guion bajo). Estos métodos también son conocidos como "métodos mágicos" o "métodos especiales". Son invocados automáticamente por Python en situaciones específicas y tienen un propósito especial en el funcionamiento de las clases.

¿Por qué se utilizan los Métodos Dunder?

Los métodos dunder permiten a los desarrolladores personalizar o modificar el comportamiento predeterminado de las clases en Python. Al implementar estos métodos en una clase, se puede cambiar cómo se comportan los objetos de esa clase en situaciones específicas, como la impresión, la comparación, la iteración, etc.

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}"
inv = Invoice('Google', 500)
print(str(inv)) # Invoice from Google for 500
```

- La clase **Invoice** tiene un constructor (**__init__**) que se ejecuta automáticamente cuando se crea una nueva instancia de la clase. Recibe dos parámetros: **client** y **total**, y los asigna como atributos de la instancia (**self.client** y **self.total**).
- La clase define un método especial Dunder llamado **__str__**. Este método es invocado cuando se utiliza la función **str()** en una instancia de la clase. Devuelve una cadena formateada que representa la factura.
- Se crea una instancia de la clase **Invoice** con el cliente "Google" y un total de 500.
- Se utiliza la función **str()** para obtener la representación en cadena de la instancia **inv**.
- La función **str()** invoca el método **__str__** de la clase **Invoice**.
- Se imprime la cadena resultante.

En resumen, este código muestra cómo definir un método Dunder especial **__str__** en una clase para controlar la representación en cadena de las instancias de esa clase. Al imprimir la instancia usando **str()**, se obtiene una cadena personalizada que representa la factura.

¿Qué es un decorador de python?

Un decorador en Python es una función especial que se utiliza para modificar o extender el comportamiento de otras funciones o métodos. Los decoradores permiten añadir funcionalidades a funciones existentes sin modificar su código interno.

¿Por qué se utilizan los Decoradores?

1. Permiten encapsular funcionalidades comunes y aplicarlas a múltiples funciones sin duplicar código.
2. Facilitan la organización del código al dividir la lógica en funciones pequeñas y agregar características adicionales mediante decoradores.

A continuación ejemplo de uso en el que el código demuestra el uso de propiedades y decoradores para crear getters y setters personalizados, así como el concepto de atributos protegidos.

```python
class Invoice:
    def __init__(self, client, total):
        self._client = client #al introducir el subrayado delante queda el dato protegido
        self._total = total
    def formatter(self):
        return f'{self._client} owes: ${self._total}'
    @property #getter para client
    def client(self):
        return self._client
    @client.setter #setter para client
    def client(self, client):
        self._client = client
    @property #getter para total
    def total(self):
        return self._total
google = Invoice('Google', 100)
print(google.client)
google.client = 'Yahoo' #cambiamos el client utilizando el setter
print(google.client)
print(google.formatter()) # Yahoo owes: $100
```

- La clase **Invoice** tiene dos atributos: **_client** y **_total**.
- Los atributos están precedidos por un guion bajo (_), lo que indica que son atributos protegidos. Esto sugiere que no deben ser accedidos directamente desde fuera de la clase, aunque en Python, esta convención depende de la buena práctica y no impide el acceso.
- **formatter** es un método que devuelve una cadena formateada representando la factura.
- Se utilizan decoradores **@property** y **@client.setter** para crear propiedades (**client** y **total**) con getters y setters personalizados.
- El getter de **client** permite acceder al valor de **_client**.
- El setter de **client** permite cambiar el valor de **_client**.
- El getter de **total** permite acceder al valor de **_total**.
- Se crea una instancia de la clase **Invoice** llamada **google**.
- Se imprime el valor de la propiedad **client** utilizando el getter.
- Se cambia el valor de la propiedad **client** utilizando el setter.
- Se vuelve a imprimir el valor de la propiedad **client**.
- **formatter** para obtener la representación de la factura después de cambiar el valor de **client**.