Programación 2

Métodos de Clase
Estándares de intercambio de datos

Métodos de clase

El decorador **@classmethod** en Python se usa para definir un método que está vinculado a la clase en sí, en lugar de a una instancia específica de la clase. Esto significa que un método de clase puede acceder y modificar el estado de la clase (como variables de clase), pero no de instancias individuales.

- **@classmethod** define un método que pertenece a la clase, no a la instancia.
- El primer argumento que recibe siempre es la **clase** (cls), no la instancia (self).
- Se utiliza cuando necesitas un método que trabaje con la clase en lugar de con instancias específicas.
- En el UML se identifica por estar subrayado (ej: -cantidadPersonas(): entero)

Métodos de clase

Uso típico de @classmethod:

- Crear **métodos de fábrica** que generen objetos de la clase.
- Trabajar con **variables de clase** en lugar de variables de instancia.
- Definir comportamientos compartidos por la clase y todas sus instancias.

Método de clase vs. Método estático

- @classmethod:
  - Recibe la clase como primer parámetro (cls).
  - Se usa cuando necesitamos trabajar con la clase en sí, como variables de clase o para métodos de fábrica.

- @staticmethod:
  - No recibe ni la clase (cls) ni la instancia (self) como argumento.
  - Se usa cuando un método no necesita acceder a la clase ni a los atributos de instancia, es simplemente un método utilitario dentro de la clase.

Métodos de clase

```python
class Persona:
    __cantidad_personas = 0  # Variable de clase

    def __init__(self, nombre:str):
        if not isinstance(nombre, str):
            raise ValueError("El nombre debe ser un string.")
        self.__nombre = nombre
        Persona.__cantidad_personas += 1

    @classmethod
    def mostrar_cantidad(cls):
        return f"Cantidad de personas: {cls.__cantidad_personas}"

class Tester:
    @staticmethod
    def test():
        # Crear algunas instancias
        p1 = Persona("Juan")
        p2 = Persona("María")
        # Llamar al método de clase sin usar una instancia específica
        print(Persona.mostrar_cantidad())  # Salida: Cantidad de personas: 2
```

Estándares de intercambio de datos - Definición

Los estándares de intercambio de datos son conjuntos de reglas y convenciones que especifican cómo los datos deben ser formateados y estructurados para que puedan ser compartidos y entendidos por diferentes sistemas o aplicaciones.

Estos estándares garantizan la interoperabilidad, es decir, la capacidad de sistemas heterogéneos para comunicarse entre sí de manera coherente.

¿Por qué los necesitamos?

Los estándares de intercambio de datos son esenciales para lograr una comunicación efectiva entre sistemas distribuidos y para evitar problemas de incompatibilidad. Al seguir estas normas, se simplifica el proceso de integración de sistemas, la automatización de tareas y la colaboración entre aplicaciones.

Serialización de Objetos

Como estamos trabajando bajo el paradigma de programación orientada a objetos, nuestros sistemas tendrán objetos conteniendo la información.

Para poder convertir esa información contenida en el objeto a un estándar de intercambio de datos necesitamos *serializarlo*.

La serialización es el proceso de convertir un objeto de un programa en una secuencia de bytes. Esto permite almacenar el objeto en un archivo, transmitirlo a través de una red o simplemente guardarlo en memoria para su posterior uso. En esencia, es como tomar una fotografía de un objeto en un momento dado y guardarla para más adelante.

Serialización/Deserialización de Objetos

¿Por qué es importante la serialización en POO?

- Persistencia: Permite guardar el estado de un objeto para su uso posterior, incluso después de que el programa se haya cerrado.
- Transmisión de datos: Facilita el envío de objetos a través de redes, por ejemplo, para comunicarse entre diferentes aplicaciones o sistemas.
- Almacenamiento: Permite guardar objetos en bases de datos o archivos de configuración.

¿Cómo funciona la serialización?

- **Serialización**/Conversión a una representación serializable: El objeto se descompone en sus componentes básicos (atributos, valores, relaciones) y se convierte en una estructura de datos que puede ser fácilmente almacenada o transmitida. (*Nosotros usaremos diccionarios y listas de diccionarios*)
- Almacenamiento o transmisión: La representación serializada se guarda en un archivo, se envía a través de una red o se almacena en memoria.
- **Deserialización**/Restauración: Cuando se necesita recuperar el objeto, la secuencia de bytes se lee y se reconstruye el objeto original a partir de ella.

Ejemplos de estándares de intercambio de datos

XML (Extensible Markup Language): Un lenguaje de marcado que permite definir estructuras de datos personalizadas. Puede utilizarse en una amplia variedad de aplicaciones y es especialmente común en la web.

JSON (JavaScript Object Notation): Un formato ligero de intercambio de datos que se utiliza ampliamente en aplicaciones web y servicios API debido a su simplicidad y facilidad de lectura y escritura por parte de humanos.

XML (Extensible Markup Language)

XML es un lenguaje de marcado que se utiliza para definir estructuras de datos personalizadas. Se basa en etiquetas que rodean los datos y permiten describir la información de manera jerárquica. A diferencia de HTML, que se utiliza para crear contenido web, XML se enfoca en la estructura y el significado de los datos en lugar de su presentación.

Ejemplos de XML

```xml
<persona>
    <nombre>Juan</nombre>
    <apellido>Pérez</apellido>
    <edad>30</edad>
</persona>
```

```xml
<libros>
    <libro>
        <titulo>El hobbit</titulo>
        <autor>J.R.R. Tolkien</autor>
        <genero>Fantasía</genero>
    </libro>
    <libro>
        <titulo>Cien años de soledad</titulo>
        <autor>Gabriel García Márquez</autor>
        <genero>Realismo mágico</genero>
    </libro>
    <libro>
        <titulo>La Odisea</titulo>
        <autor>Homero</autor>
        <genero>Epopeya</genero>
    </libro>
</libros>
```

Características

Jerarquía: Los datos se organizan en una estructura jerárquica de elementos que pueden contener subelementos.

Etiquetas: Cada dato se encierra en etiquetas que describen su tipo o significado.

Extensible: Permite definir sus propias etiquetas y estructuras de datos personalizadas, lo que lo hace altamente adaptable.

Legible por Humanos: XML es legible por humanos y, por lo tanto, es útil en situaciones donde la claridad y la comprensión son importantes.

XML en python

En Python podemos trabajar con XML utilizando una serie de módulos y bibliotecas que facilitan la lectura, manipulación y generación de documentos XML. Uno de los módulos más comunes para trabajar con XML en Python es `xml.etree.ElementTree`.

Ejemplo de uso

```
import xml.etree.ElementTree as ET

class Persona:
    def __init__(self, dni:int, nombre:str, apellido:str):
        if not isinstance(dni, int) or dni < 0:
            raise ValueError("El DNI debe ser un entero positivo.")
        if not isinstance(nombre, str) :
            raise ValueError("El nombre debe ser un string.")
        if not isinstance(apellido, str):
            raise ValueError("El apellido debe ser un string.")
        self.__dni = dni
        self.__nombre = nombre
        self.__apellido = apellido
```

Deserialización/reconstrucción

```python
@classmethod
def from_xml(cls, xml_data) -> "Persona":
    root = ET.fromstring(xml_data)
    # Validar si los nodos existen en el XML
    dni = root.find("dni").text if root.find("dni") is not None else '0'
    nombre = root.find("nombre").text if root.find("nombre") is not None else ''
    apellido = root.find("apellido").text if root.find("apellido") is not None else ''
    return cls(int(dni), nombre, apellido)
```

Serialización

```python
def to_xml(self)->str:
    root = ET.Element("persona")
    ET.SubElement(root, "dni").text = str(self.__dni)
    ET.SubElement(root, "nombre").text = self.__nombre
    ET.SubElement(root, "apellido").text = self.__apellido
    return ET.tostring(root, encoding="utf-8").decode("utf-8")
```

```python
def __str__(self):
    return f"DNI: {self.__dni}, Nombre: {self.__nombre}, Apellido: {self.__apellido}"
```

```python
class Tester:
    @staticmethod
    def test_xml():
        # Lista de diccionarios con información
        datos = [
            {"dni": "12345678", "nombre": "Juan", "apellido": "Pérez"},
            {"dni": "98765432", "nombre": "María", "apellido": "Gómez"},
            {"dni": "45678901", "nombre": "Pedro", "apellido": "Rodríguez"}
        ]

        # Crear lista de objetos Persona
        lista_personas = []
        for dicc in datos:
            persona = Persona(int(dicc["dni"]), dicc["nombre"], dicc["apellido"])
            lista_personas.append(persona)

        # Mostrar cada objeto persona
        for obj_persona in lista_personas:
            print(obj_persona)
```

```python
# Crear XML de cada persona y guardarlo en un archivo
with open("personas.xml", "w", encoding="utf-8") as archivo:
    for obj_persona in lista_personas:
        archivo.write(str(obj_persona.to_xml()) + "\n")

print("\nDatos guardados en personas.xml")

# Leer XML desde el archivo y recrear los objetos Persona
lista_personas_desde_xml = []
with open("personas.xml", "r", encoding="utf-8") as archivo:
    for linea in archivo:
        persona = Persona.from_xml(linea.strip())
        lista_personas_desde_xml.append(persona)

# Mostrar objetos recreados desde el XML
print("\nObjetos recreados desde XML:")
for obj_persona in lista_personas_desde_xml:
    print(obj_persona)
```

XML: ventajas

- Legibilidad: Fácil para humanos de leer y escribir.
- Flexibilidad: Permite definir etiquetas personalizadas.
- Compatibilidad: Ampliamente compatible con diferentes plataformas y lenguajes.

XML: desventajas

Verboso: XML tiende a ser más verboso en comparación con otros formatos de intercambio de datos como JSON. Esto significa que puede requerir más caracteres para representar la misma información.

Consumo de ancho de banda y espacio en disco: Debido a su naturaleza verbosa, los documentos XML pueden ocupar más espacio en disco y requerir más ancho de banda para la transferencia de datos, lo que puede ser un problema en situaciones donde los recursos son limitados.

Procesamiento más lento: Analizar documentos XML puede ser más lento en comparación con otros formatos más compactos, lo que puede afectar el rendimiento en aplicaciones que manejan grandes cantidades de datos.

No es tan eficiente para datos pequeños: Para pequeñas cantidades de datos, XML puede ser excesivo y no tan eficiente en términos de espacio y tiempo de procesamiento.

No es adecuado para todas las situaciones: Aunque XML es ampliamente aplicable, no es la mejor opción para todas las situaciones. Por ejemplo, en aplicaciones web modernas, donde la eficiencia en la transmisión de datos es esencial, JSON es a menudo preferido sobre XML.

JSON (JavaScript Object Notation)

JSON es un formato ligero de intercambio de datos basado en texto que se utiliza para representar objetos y estructuras de datos. Aunque se originó en el contexto de JavaScript, se ha convertido en un estándar ampliamente adoptado y se utiliza en una variedad de lenguajes de programación.

Formato / composición

JSON está constituído por dos estructuras:

A. Una colección de pares de `"clave" : valor`. Dependiendo del lenguaje esto es conocido como un **objeto**, registro, estructura, **diccionario**, tabla hash, lista de claves o un arreglo asociativo.

B. Una lista ordenada de elementos. En la mayoría de los lenguajes, esto se implementa como arreglos, vectores, listas o secuencias. (Ej: lista de diccionarios)

Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

Características

Sintaxis Ligera: JSON utiliza una sintaxis simple y concisa para representar datos.

Estructuras de Datos Simples: Soporta tipos de datos como objetos, arreglos, cadenas, números, booleanos y valores nulos.

Facilidad de Lectura y Escritura: Es fácil de entender para los humanos y fácil de analizar y generar en aplicaciones.

Amplia Adopción: Se utiliza en servicios web, APIs y aplicaciones en línea debido a su eficiencia y simplicidad.

Ejemplo de JSON

```
[
    {
        "nombre": "Juan",
        "apellido": "Perez",
        "edad": 30,
        "direccion": {
            "calle": "Av. Siempre Viva",
            "numero": 742
        }
    }
]
```

Trabajando con JSON en python

Para trabajar con JSON simplemente debemos importar la librería que python trae incorporada para codificar y decodificar archivos .json:

```
import json
```

La librería tiene dos métodos principales para convertir a/desde string:

- **json.dumps(...)**: Convierte objetos nativos de Python en cadenas JSON. (dumps es abreviatura de '*dump to string*'). Lo usaremos para serializar un objeto.
- **json.loads(...)**: Convierte cadenas JSON en objetos Python.
(loads es la abreviación de '*load string')*. Lo usaremos en el método para Reconstruir un objeto con información de un string JSON. (Deserializacion)

Ejemplo de uso básico

```python
import json

# Convertir diccionario Python a JSON
persona = {"nombre": "Juan", "edad": 30, "ciudad": "Madrid"}
json_data_str = json.dumps(persona)
print(json_data_str)  # {"nombre": "Juan", "edad": 30, "ciudad": "Madrid"}

# Convertir JSON a diccionario Python
persona_dicc = json.loads(json_data_str)
print(persona_dicc["nombre"])  # Juan
```

Serialización de Objetos

```python
import json

class Persona:
    def __init__(self, dni:int, nombre:str, apellido:str):
        if not isinstance(dni, int):
            raise ValueError("El DNI debe ser un entero.")
        if not isinstance(nombre, str) :
            raise ValueError("El nombre debe ser un string.")
        if not isinstance(apellido, str):
            raise ValueError("El apellido debe ser un string.")
        self.__dni = dni
        self.__nombre = nombre
        self.__apellido = apellido

    def to_json(self):
        dicc_persona = {"dni": self.__dni, "nombre": self.__nombre, "apellido": self.__apellido}
        return json.dumps(dicc_persona, ensure_ascii=False)
```

```python
import json

class Persona:
    def __init__(self, dni:int, nombre:str, apellido:str):
        if not isinstance(dni, int):
            raise ValueError("El DNI debe ser un entero.")
        if not isinstance(nombre, str) :
            raise ValueError("El nombre debe ser un string.")
        if not isinstance(apellido, str):
            raise ValueError("El apellido debe ser un string.")
        self.__dni = dni
        self.__nombre = nombre
        self.__apellido = apellido

    def to_json(self):
        dicc_persona = {"dni": self.__dni, "nombre": self.__nombre, "apellido": self.__apellido}
        return json.dumps(dicc_persona, ensure_ascii=False)

    @classmethod
    def from_json(cls, json_data):
        datos = json.loads(json_data)
        return cls(datos["dni"], datos["nombre"], datos["apellido"])
```

Serialización

Deserialización/reconstrucción

Leyendo un archivo json

Tenemos la función **`json.load(archivo)`** (sin la ‘s’ final) que carga los datos contenidos en el archivo que previamente tenemos que abrir.

```
with open('datos.json') as json_file:
    data = json.load(json_file)
    ...
```

JSON: ventajas

- **Ligereza**: JSON es conocido por ser un formato de intercambio de datos ligero. Esto significa que los datos en formato JSON tienden a ocupar menos espacio de almacenamiento y ancho de banda en comparación con otros formatos como XML. Esta ventaja es especialmente relevante en aplicaciones web y móviles donde la eficiencia de los recursos es crucial.
- **Facilidad de uso**: JSON es fácil de entender y utilizar tanto para los desarrolladores como para las máquinas. Los datos se representan en pares clave-valor, lo que facilita la identificación de los datos y su acceso. Esto lo hace ideal para el intercambio de datos entre aplicaciones y servicios web.
- **Integración**: JSON es altamente compatible con la mayoría de los lenguajes de programación y es ampliamente admitido en la comunidad de desarrollo. Esto significa que puedes utilizar JSON para intercambiar datos entre diferentes componentes de software sin problemas de incompatibilidad.

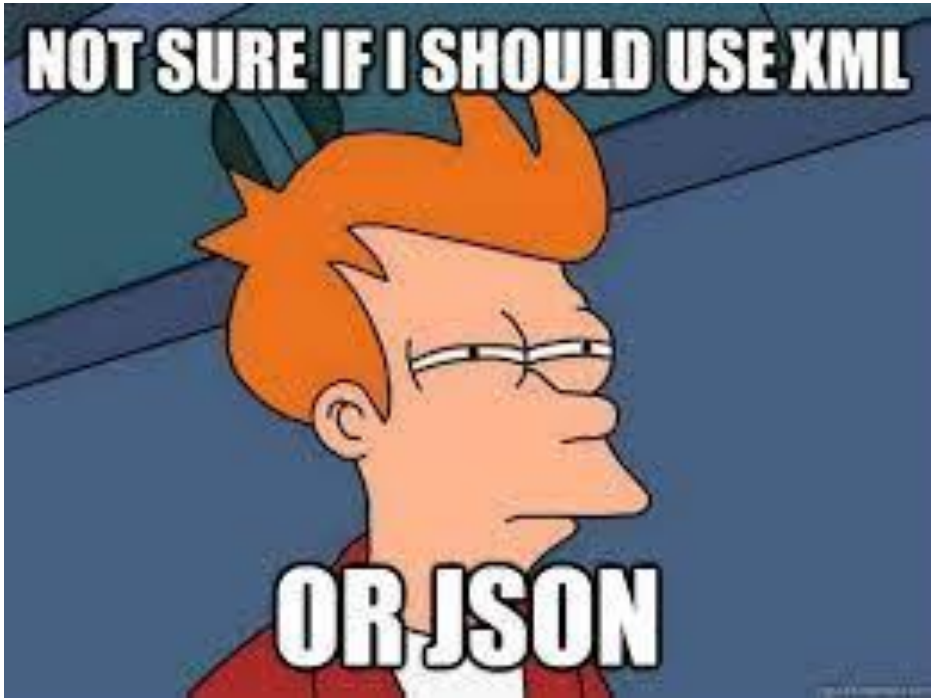
JSON: desventajas

- **Menos estructurado**: Aunque JSON es eficiente para representar datos semiestructurados y objetos simples, puede no ser la mejor opción cuando se trata de datos altamente estructurados o documentos complejos. XML, en este sentido, ofrece una estructura más rica y jerárquica que puede ser más adecuada para ciertos tipos de datos.
- **Menos legible para humanos**: Aunque JSON es relativamente legible para humanos en comparación con otros formatos binarios, como el protocolo de transferencia de hipertexto (HTTP), su sintaxis es más concisa y puede ser menos intuitiva de leer en comparación con XML. Esto puede dificultar la depuración y la edición manual de datos JSON.

Elección del estándar

La elección entre XML y JSON debe basarse en las necesidades específicas del proyecto. XML puede ser preferible cuando se necesita una estructura jerárquica y una representación más formal de datos. En cambio, JSON es ideal para casos en los que la eficiencia y la facilidad de uso son prioritarias, como en aplicaciones web y móviles. Ejemplos:

- XML se utiliza a menudo en configuraciones de intercambio de datos estructurados, como la configuración de aplicaciones y la representación de documentos.
- JSON es comúnmente utilizado en el desarrollo web y de aplicaciones móviles para la comunicación entre el cliente y el servidor debido a su ligereza y facilidad de uso.



JSON

```
{
"siblings": [
{"firstName":"Anna","lastName":"Clayton"},
{"firstName":"Alex","lastName":"Clayton"}
]
}
```

XML

```
<siblings>
<sibling>
<firstName>Anna</firstName>
<lastName>Clayton</lastName>
</sibling>
<sibling>
<firstName>Alex</firstName>
<lastName>Clayton</lastName>
</sibling>
</siblings>
```


WHEN THIRD PARTY API
SENDS XML INSTEAD OF JSON
imgflip.com