Creado por:

Isabel Maniega

JSON / XML con Python

Módulo Json

JSON se ha convertido en el estándar por defecto para el intercambio de información. Sus usos son:

- Transportar datos Tal vez esté
- Recopilando información a través de una API
- Almacenando sus datos en una base de datos de documentos.

Las siglas vienen de:

- **J**ava...
- ...**S**cript
- **O**bject
- **N**otation

JSON es la respuesta a una necesidad bastante básica: la necesidad de transferir datos que son el contenido de un objeto o un conjunto de objetos. El mecanismo que la soluciona debe ser portable e independiente de la plataforma.

¿Cómo podemos satisfacer tal requisito?

El problema con el que nos enfrentamos es cómo representar un objeto (entendido como un conjunto de datos de diferentes tipos, incluidos otros objetos) o incluso un único valor de forma que pueda sobrevivir a las transferencias de red y las conversiones entre plataformas.

JSON resuelve el problema con dos trucos simples:

- utiliza texto codificado en UTF-8, lo que significa que no se utilizan formatos dependientes de la máquina o la plataforma; también significa que los datos que lleva JSON son legibles (mal, pero siempre legibles) y comprensibles para los humanos;
- utiliza un formato simple y no muy expandido (podemos llamarlo sintaxis, o incluso gramática) para representar dependencias mutuas y relaciones entre diferentes partes de los objetos, y es capaz de transferir no solo los valores de las propiedades de los objetos, sino también sus nombres.

En JSON, puede ser un valor sin nombre como un número, una cadena, un booleano o… nada, aunque esto no es lo que más nos gusta de JSON. JSON es capaz de transportar datos mucho más complejos, recopilados y agregados en compuestos más grandes.

Si desea transferir no solo datos sin procesar, sino también todos los nombres asociados a ellos (como la forma en que los objetos mantienen sus propiedades), JSON ofrece una sintaxis que parece un pariente cercano del diccionario de Python, que es, de hecho, un conjunto de pares `clave:valor`. Hacer tal suposición nos lleva a la siguiente pregunta: ¿podemos usar la sintaxis de Python para codificar y decodificar mensajes de red en REST?

Ejemplo de Json:

```
{

    "firstName": "Jasmine",

    "lastName": "Doe",

    "hobbies": ["running", "cooking", "singing"],

    "age": 35,

    "children": [

        {
            "firstName": "Alice",

            "age": 6
        },
        {
            "firstName": "Bob",

            "age": 8
        }
    ]

}
```

Para poder manejar estos datos usaremos el modulo **json:**

```
import json
```

Concepto básico:

- **Serialización:** Consiste en convertir un objeto de Python (normalmente una lista o diccionario) en un string.
- **Deserialización:** Consiste en convertir un string en un objeto de Python (normalmente una lista o diccionario).

In [1]:
```
# Ejemplo de serialización

import json

data = {
    "presidente": {
```

```
        "nombre": "Zaphod Beeblebrox",
        "especie": "Betelgeusian"
    }
}

json_string = json.dumps(data)
json_string
```

Out[1]:
```
'{"presidente": {"nombre": "Zaphod Beeblebrox", "especie": "Betelgeusia
n"}}'
```

In [2]:
```
type(json_string)
```

Out[2]:
```
str
```

In [3]:
```
json_string["presidente"]

# TypeError: string indices must be integers
# No podemos recorrerlo para ello será necesario decodificarlo
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call las
t)
Cell In[3], line 1
----> 1 json_string["presidente"]
      3 # TypeError: string indices must be integers
      4 # No podemos recorrerlo para ello será necesario decodificarlo

TypeError: string indices must be integers
```

In [4]:
```
# La opción de dumps nos sirve para escribirlo en un archivo de formato J

with open("archivo_data.json", "w") as write_file:
    json.dump(data, write_file)
```

In [5]:
```
# Indent nos permite añadir un salto a cada componente del Json:

with open("archivo_data.json", "w") as write_file:
    json.dump(data, write_file, indent=2)
```

In [12]:
```
# sort_keys= True: la salida de los diccionarios se ordenará por clave:
# Salida esperada: {"age": 30, "city": "New York", "name": "John"}

data = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

with open("archivo_data.json", "w") as write_file:
    json.dump(data, write_file, indent=2, sort_keys=True)
```

In [7]:
```
# Ejemplo de deserialización
# Convertir datos codificados en JSON en objetos de Python.

json_decoded = json.loads(json_string)
json_decoded
```

```
Out[7]: {'presidente': {'nombre': 'Zaphod Beeblebrox', 'especie': 'Betelgeusia
        n'}}
```

```
In [8]: type(json_decoded)

        # Con loads podemos recorrer el diccionario y extraer la información.
```

```
Out[8]: dict
```

```
In [9]: json_decoded['presidente']
```

```
Out[9]: {'nombre': 'Zaphod Beeblebrox', 'especie': 'Betelgeusian'}
```

```
In [10]: json_decoded['presidente']['nombre']
```

```
Out[10]: 'Zaphod Beeblebrox'
```

Como puede ver, Python utiliza un pequeño conjunto de reglas simples para crear mensajes JSON a partir de sus datos nativos. Aquí está:

Python data	JSON element
`dict`	`object`
`list or tuple`	`array`
`string`	`string`
`int or float`	`number`
`True / False`	`true / false`
`None`	`null`

Pero hasta ahora parece simple y coherente, pero ¿dónde está la trampa?

La trampa está aquí:

```
import json

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


some_man = Who('John Doe', 42)
print(json.dumps(some_man))
```

El resultado que verás es extremadamente decepcionante:

```
TypeError: Object of type 'Class' is not JSON serializable
```

Sí, esa es la clave. No puedes simplemente volcar el contenido de un objeto, incluso un objeto tan simple como este.

Por supuesto, si no necesitas nada más que un conjunto de propiedades de objeto y sus valores, puedes realizar un truco (un poco sucio) y volcar no el objeto en sí, sino el contenido de su propiedad `__dict__`. Funcionará, pero esperamos más.

¿Qué deberíamos hacer?

Un enfoque se basa en el hecho de que la serialización se realiza mediante el método denominado `default()`, que forma parte de la clase `json.JSONEncoder`. Te da la oportunidad de sobrecargar el método que define una subclase de JSONEncoder y pasarlo a `dumps()`.

Pero, ¿qué sucede con los objetos de Python? ¿Podemos deserializarlos de la misma manera que realizamos la serialización?

Como probablemente esperes, deserializar un objeto puede requerir algunos pasos adicionales. Sí, de hecho. Como `loads()` no puede adivinar qué objeto (de qué clase) necesitas deserializar, debes proporcionar esta información.

Observa el fragmento que hemos proporcionado en la ventana del editor.

Como puedes ver, hay un argumento de palabra clave llamado `object_hook`, que se usa para señalar la función responsable de crear un objeto nuevo de una clase necesaria y de llenarlo con datos reales.

Nota: la función `decode_who()` recibe una entidad de Python, o más específicamente, un diccionario. Como el constructor de Who espera dos valores ordinarios, una cadena y un número, no un diccionario, tenemos que usar un pequeño truco: hemos empleado el operador doble `*` para convertir el directorio en una lista de argumentos de palabras clave construida a partir de los pares `key:value` del diccionario. Por supuesto, las claves del diccionario deben tener los mismos nombres que los parámetros del constructor.

Nota: la función, especificada por `object_hook` se invocará solo cuando la cadena JSON describa un objeto JSON. Lo sentimos, no hay excepciones a esta regla.

In [13]:
```
import json


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


def encode_who(w):
    if isinstance(w, Who):
        return w.__dict__
    else:
        raise TypeError(w.__class__.__name__ + 'is not JSON serializable'


def decode_who(w):
    return Who(w['name'], w['age'])
```

```
old_man = Who("Jane Doe", 23)
json_str = json.dumps(old_man, default=encode_who)
new_man = json.loads(json_str, object_hook=decode_who)
print(type(new_man))
print(new_man.__dict__)
```

```
<class '__main__.Who'>
{'name': 'Jane Doe', 'age': 23}
```

Como ya se ha dicho, también es posible utilizar un enfoque más puro basado en objetos, que se basa en la redefinición de la clase JSONDecoder. Lamentablemente, esta variante es más complicada que su contraparte de codificación.

No necesitamos reescribir ningún método, pero sí tenemos que redefinir el constructor de la superclase, lo que hace que nuestro trabajo sea un poco más laborioso. El nuevo constructor tiene como objetivo hacer solo una cosa: establecer una función para la creación de objetos.

Como puede ver, esto es exactamente lo mismo que hicimos antes, pero expresado en un nivel diferente.

Nos complace informarle que ahora hemos reunido suficiente conocimiento para pasar al siguiente nivel. Volveremos a tratar algunos problemas de red, pero también queremos mostrarle algunas herramientas nuevas y útiles.

In [14]:
```
import json


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


class MyEncoder(json.JSONEncoder):
    def default(self, w):
        if isinstance(w, Who):
            return w.__dict__
        else:
            return super().default(self, z)


class MyDecoder(json.JSONDecoder):
    def __init__(self):
        json.JSONDecoder.__init__(self, object_hook=self.decode_who)

    def decode_who(self, d):
        return Who(**d)


some_man = Who('Jane Doe', 23)
json_str = json.dumps(some_man, cls=MyEncoder)
new_man = json.loads(json_str, cls=MyDecoder)
```

```
print(type(new_man))
print(new_man.__dict__)
```

```
<class '__main__.Who'>
{'name': 'Jane Doe', 'age': 23}
```

Archivos XML

Procesamiento de XML en Python

Python se utiliza habitualmente para procesar distintos tipos de datos. Tal vez, mientras trabaja como programador, tenga que leer o crear un archivo de datos en formato XML. Pronto, hacerlo será pan comido.

La biblioteca estándar de Python ofrece algunos módulos interesantes para trabajar con XML:

- xml.etree.ElementTree: tiene una API muy simple para analizar y crear datos XML. Es una excelente opción para personas que nunca han trabajado con el Modelo de objetos de documento (DOM) antes.
- xml.dom.minidom: es la implementación mínima del Modelo de objetos de documento (DOM). Al utilizar el DOM, el enfoque para un documento XML es ligeramente diferente, porque se analiza en una estructura de árbol en la que cada nodo es un objeto.
- xml.sax: SAX es un acrónimo de “Simple API for XML”. SAX es una interfaz en Python para el análisis de documentos XML impulsado por eventos. A diferencia de los módulos anteriores, analizar un documento XML simple utilizando este módulo requiere más trabajo.

En este curso aprenderás a crear y procesar documentos XML utilizando el módulo `xml.etree.ElementTree`. No perdamos más tiempo. ¡Vamos allá!

¿Qué es XML?

Extensible Markup Language (XML) es un lenguaje de marcado diseñado para almacenar y transportar datos, por ejemplo, mediante sistemas que utilizan el protocolo de comunicación SOAP. Una de sus principales ventajas es la capacidad de definir sus propias etiquetas que hacen que el documento sea más legible para los humanos. XML es un estándar recomendado por la organización W3C. Veamos qué elementos contienen los documentos XML:

- prólogo: la primera línea (opcional) del documento. En el prólogo, puede especificar la codificación de caracteres, por ejemplo, `<?xml version="1.0" encoding="ISO-8859-2"?>` cambia la codificación de caracteres predeterminada (UTF-8) a ISO-8859-2.
- elemento raíz: el documento XML debe tener un elemento raíz que contenga todos los demás elementos. En el ejemplo siguiente, el elemento principal es la etiqueta `data`.

- elementos: consisten en etiquetas de apertura y cierre. Los elementos incluyen texto, atributos y otros elementos secundarios. En el ejemplo siguiente, podemos encontrar el elemento book con el atributo title y dos elementos secundarios (`author` y `year`).
- atributos: estos se colocan en las etiquetas de apertura. Consisten en pares clave-valor, p. ej., `title = "El Principito"` .

NOTA: Cada etiqueta XML abierta debe tener una etiqueta `closing` correspondiente.

```
<?xml version="1.0"?>
<data>
    <book title="The Little Prince">
        <author>Antoine de Saint-Exupéry</author>
        <year>1943</year>
    </book>
    <book title="Hamlet">
        <author>William Shakespeare</author>
        <year>1603</year>
    </book>
</data>
```

Análisis de XML (parte 1)

Procesar archivos XML en Python es muy fácil gracias a la clase `ElementTree` proporcionada por el módulo `xml.etree.ElementTree` . El objeto `ElementTree` es responsable de presentar el documento XML en forma de árbol en el que podemos movernos hacia arriba o hacia abajo.

Primero, necesitamos importar el módulo apropiado y definir un alias para él. Es común usar el alias ET, pero por supuesto puedes darle el nombre que quieras. Para crear un árbol (un objeto `ElementTree`) a partir de un documento XML existente, pásalo al método parse de la siguiente manera:

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
```

El método `getroot` retorna el elemento raíz. Con acceso al elemento raíz, podemos llegar a cualquier elemento del documento. Cada uno de estos elementos está representado por una clase llamada Element.

Además del método parse, podemos utilizar el método llamado `fromstring` , que, como argumento, toma XML como cadena:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(your_xml_as_string)
```

NOTA: El método `fromstring` no devuelve el objeto `ElementTree` , sino que devuelve el elemento raíz representado por la clase `Element` .

XML parsing (part 2)

Ya sabes cómo acceder al elemento raíz. Vamos a usarlo para visitar los elementos de nuestro árbol: mira el código en el editor.

Resultado:

```
The root tag is: data
The root has the following children:
book {'title': 'The Little Prince'}
book {'title': 'Hamlet'}
```

El elemento raíz y todos sus elementos secundarios son objetos `Element`. En el ejemplo anterior, utilizamos las siguientes propiedades:

- tag: devuelve el nombre de la etiqueta como una cadena
- attrib: devuelve todos los atributos de la etiqueta como un diccionario. Para recuperar el valor de un solo atributo, utilice su clave, p. ej., `child.attrib['title']`.

In [15]:
```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
print('The root tag is:', root.tag)
print('The root has the following children:')
for child in root:
    print(child.tag, child.attrib)
```

```
The root tag is: data
The root has the following children:
book {'title': 'The Little Prince'}
book {'title': 'Hamlet'}
```

Análisis XML (parte 3)

Además de iterar sobre los elementos del árbol, podemos acceder a ellos directamente mediante índices. Observa el ejemplo siguiente en el que utilizamos el elemento book actual para recuperar texto de sus elementos secundarios. Observa el código en el editor.

Resultado:

```
My books:

Title:  The Little Prince
Author: Antoine de Saint-Exupéry
Year:  1943

Title:  Hamlet
Author: William Shakespeare
Year:  1603
```

Durante cada iteración, hacemos referencia a los elementos secundarios del elemento book mediante índices. El índice `0` hace referencia al primer elemento secundario del elemento book, mientras que el índice `1` hace referencia a su segundo elemento secundario. La visualización de texto es posible gracias a la propiedad `text`, disponible en el objeto `Element`.

NOTA: Los índices también se utilizan en anidamientos más profundos, por ejemplo, `root [0] [0]` .text devuelve el primer elemento del libro y luego muestra el texto ubicado en su primer elemento secundario.

In [16]:
```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
print("My books:\n")
for book in root:
    print('Title: ', book.attrib['title'])
    print('Author:', book[0].text)
    print('Year: ', book[1].text, '\n')
```

```
My books:

Title:  The Little Prince
Author: Antoine de Saint-Exupéry
Year:  1943

Title:  Hamlet
Author: William Shakespeare
Year:  1603
```

Análisis de XML (parte 4)

El módulo `xml.etree.ElementTree`, o más precisamente, la clase `Element`, proporciona varios métodos útiles para buscar elementos en un documento XML. Comencemos con el método llamado `iter`.

El método `iter` devuelve todos los elementos al pasar la etiqueta como argumento. El elemento que lo llama se trata como el elemento principal desde el que comienza la búsqueda. Para encontrar todas las coincidencias, el iterador itera recursivamente a través de todos los elementos secundarios y sus elementos anidados.

Observe el código en el editor para ver un ejemplo de una búsqueda de todos los elementos con la etiqueta `author`.

Resultado:

```
Antoine de Saint-Exupéry
William Shakespeare
```

In [17]:
```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
```

```
for author in root.iter('author'):
    print(author.text)
```

```
Antoine de Saint-Exupéry
William Shakespeare
```

Análisis XML (parte 5)

El objeto Element tiene un método llamado `findall` para buscar elementos secundarios directos. A diferencia del método `iter`, el método `findall` solo busca los elementos secundarios en el primer nivel de anidación. Eche un vistazo al ejemplo en el editor.

El ejemplo no devuelve ningún resultado, porque el método `findall` solo puede iterar sobre los elementos del libro que son los elementos secundarios más cercanos del elemento raíz. El código correcto se ve así:

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for book in root.findall('book'):
    print(book.get('title'))
```

Resultado:

```
The Little Prince
Hamlet
```

Para visualizar el valor de los atributos `title`, utilizamos el método `get`, que luce mucho mejor que una llamada `book.attrib['title']`. Su uso es muy sencillo: solo hay que introducir el nombre del atributo y opcionalmente (como segundo argumento) el valor que se devolverá si no se encuentra el atributo (por defecto es None).

NOTA: El método `findall` también acepta una expresión XPath. Te animamos a que profundices en el conocimiento de las expresiones XPath y lo apliques al ejemplo mostrado.

In [18]:
```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for book in root.findall('book'):
    print(book.get('title'))
```

```
The Little Prince
Hamlet
```

Análisis de XML (parte 6)

Otro método útil que se utiliza para analizar un documento XML es un método llamado `find`. El método `find` devuelve el primer elemento secundario que contiene la

etiqueta especificada o la expresión XPath correspondiente. Observa el código en el editor.

Resultado:

```
The Little Prince
```

En el ejemplo, utilizamos el método `find` para encontrar el primer elemento secundario que contiene la etiqueta book y luego mostramos el valor de su atributo `title`. Ten en cuenta que el elemento desde el que iniciamos la búsqueda es el elemento raíz.

In [19]:
```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
print(root.find('book').get('title'))
```

```
The Little Prince
```

Modificación de un documento XML (parte 1)

Ya aprendiste a analizar un documento XML. Es hora de dar el siguiente paso. Modifiquemos el árbol de elementos y creemos un nuevo archivo XML basado en él con los siguientes datos de la película:

```
<?xml version="1.0"?>
<data>
    <movie title="The Little Prince" rate="5"></movie>
    <movie title="Hamlet" rate="5"></movie>
</data>
```

¿Te preguntas si será difícil convertir datos de libros en datos de películas? Gracias al módulo `xml.etree.ElementTree`, es muy fácil.

Para cambiar la etiqueta del objeto Element, debemos asignar un nuevo valor a su propiedad tag. Observa el código en el editor.

Resultado:

```
movie {'title': 'The Little Prince'}
author : Antoine de Saint-Exupéry
year : 1943
movie {'title': 'Hamlet'}
author : William Shakespeare
year : 1603
```

En el ejemplo, durante cada iteración a través de los elementos `book`, los reemplazamos con la etiqueta `movie` y luego nos aseguramos de que todos los cambios se hayan realizado correctamente.

In [20]:
```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for child in root:
    child.tag = 'movie'
    print(child.tag, child.attrib)
    for sub_child in child:
        print(sub_child.tag, ':', sub_child.text)
```

```
movie {'title': 'The Little Prince'}
author : Antoine de Saint-Exupéry
year : 1943
movie {'title': 'Hamlet'}
author : William Shakespeare
year : 1603
```

Modificación de un documento XML (parte 2)

Nuestro XML tiene algunos elementos innecesarios, como el autor y el año. Para eliminarlos, necesitamos utilizar el método llamado `remove`, proporcionado por la clase `Element`. Observa el código en el editor.

Resultado:

```
movie {'title': 'The Little Prince'}
movie {'title': 'Hamlet'}
```

El método remove elimina el elemento secundario que se pasa como argumento, que debe ser un objeto `Element`. Ten en cuenta que para este propósito usamos el método llamado `find`, con el que ya estás familiarizado.

In [21]:
```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for child in root:
    child.tag = 'movie'
    child.remove(child.find('author'))
    child.remove(child.find('year'))
    print(child.tag, child.attrib)
    for sub_child in child:
        print(sub_child.tag, ':', sub_child.text)
```

```
movie {'title': 'The Little Prince'}
movie {'title': 'Hamlet'}
```

Modificación de un documento XML (parte 3)

¿Recuerdas el método get que obtiene el valor del atributo? El objeto `Element` también tiene un método llamado `set`, que permite configurar cualquier atributo. Observa el código en el editor.

Resultado:

```
movie {'title': 'The Little Prince', 'rate': '5'}
movie {'title': 'Hamlet', 'rate': '5'}
```

El método set toma como argumentos el atributo `nombre` y su `valor`. En nuestro caso, lo utilizamos para establecer la calificación más alta para cada una de las películas.

In [22]:
```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for child in root:
    child.tag = 'movie'
    child.remove(child.find('author'))
    child.remove(child.find('year'))
    child.set('rate', '5')
    print(child.tag, child.attrib)
    for sub_child in child:
        print(sub_child.tag, ':', sub_child.text)
```

```
movie {'title': 'The Little Prince', 'rate': '5'}
movie {'title': 'Hamlet', 'rate': '5'}
```

Modificación de un documento XML (parte 4)

Debes haber notado que el documento XML modificado no se guarda en ningún lugar. Para guardar todos los cambios que hemos realizado en el árbol, tenemos que utilizar el método llamado `write`.

El método `write` tiene un solo argumento obligatorio, que es un nombre de archivo del archivo XML de salida, o un objeto de archivo abierto para escritura. Además, podemos definir la codificación de caracteres utilizando el segundo argumento (el valor predeterminado es US-ASCII). Para agregar un prólogo a nuestro documento, debemos pasar `True` en el tercer argumento.

Observa el ejemplo en el editor, en el que guardamos el árbol modificado en un archivo llamado `movies.xml`.

El archivo creado se ve así:

```
<?xml version='1.0' encoding='UTF-8'?>
<data><movie rate="5" title="The Little Prince" /><movie
rate="5" title="Hamlet" /></data>
```

In [23]:
```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for child in root:
    child.tag = 'movie'
    child.remove(child.find('author'))
    child.remove(child.find('year'))
    child.set('rate', '5')
    print(child.tag, child.attrib)
    for sub_child in child:
```

```
        print(sub_child.tag, ':', sub_child.text)

tree.write('movies.xml', 'UTF-8', True)
```

```
movie {'title': 'The Little Prince', 'rate': '5'}
movie {'title': 'Hamlet', 'rate': '5'}
```

Creación de un documento XML (parte 1)

Durante este curso, no ha tenido la oportunidad de crear un objeto `Element`. Veamos cómo crear un documento XML en Python.

El constructor de la clase `Element` toma dos argumentos. El primero es el nombre de la etiqueta que se creará, mientras que el segundo (opcional) es el diccionario de atributos. En el ejemplo del editor, hemos creado el elemento raíz representado por una etiqueta de datos sin atributos; observe el código.

Resultado:

```
<data />
```

En el ejemplo anterior, utilizamos el método `dump`, que nos permite depurar todo el árbol o un solo elemento.

In [24]:
```
import xml.etree.ElementTree as ET

root = ET.Element('data')
ET.dump(root)
```

```
<data />
```

Creación de un documento XML (parte 2)

Además de la clase `Element`, el módulo `xml.etree.ElementTree` ofrece una función para crear elementos secundarios llamada `SubElement`. La función `SubElement` toma tres argumentos.

El primero define el elemento principal, el segundo define el nombre de la `etiqueta` y el tercero (opcional) define los atributos del elemento. Veamos cómo se ve en acción y analicemos el código en el editor.

Resultado:

```
<data><movie rate="5" title="The Little Prince" /><movie
rate="5" title="Hamlet" /></data>
```

En el ejemplo, hemos añadido dos elementos secundarios llamados movie al elemento `root`. Los elementos creados son objetos de la clase `Element`, por lo que podemos utilizar todos los métodos que aprendimos durante este curso.

NOTA: Para guardar un documento utilizando el método `write`, necesitamos tener un objeto `ElementTree`. Para ello, pasa nuestro elemento raíz a su constructor:

```
tree = ET.ElementTree(root)
```

In [25]:
```
import xml.etree.ElementTree as ET

root = ET.Element('data')
movie_1 = ET.SubElement(root, 'movie', {'title': 'The Little Prince', 'ra
movie_2 = ET.SubElement(root, 'movie', {'title': 'Hamlet', 'rate': '5'})
ET.dump(root)
```

```
<data><movie title="The Little Prince" rate="5" /><movie title="Hamlet" ra
te="5" /></data>
```

Creado por:

Isabel Maniega