1.3_Gestion archivos

June 13, 2025

Creado por:

Isabel Maniega

0.0.1 1. Comprender los formatos de archivo en la adquisición de datos

1. **CSV (valores separados por comas)** Atributos clave

Los archivos CSV almacenan datos tabulares en formato de texto sin formato con la extensión .csv. Un archivo típico contiene valores separados por comas, pero también se permiten otros separadores como punto y coma o tabulación. Se debe enfatizar que solo se puede usar un tipo de separador en un archivo CSV.

Es simple y se usa ampliamente por su compatibilidad con muchas aplicaciones. Carece de soporte para tipos de datos y puede ser ineficiente para grandes conjuntos de datos. Los archivos CSV se usan comúnmente para intercambiar datos tabulares entre diferentes sistemas o aplicaciones. Son populares en escenarios donde la interoperabilidad y la simplicidad son más importantes que las funciones avanzadas.

Los archivos CSV son fácilmente legibles tanto por humanos como por máquinas. Se usan para tareas como importar y exportar datos desde bases de datos, hojas de cálculo y herramientas de análisis. Sin embargo, es posible que no sean la mejor opción para grandes conjuntos de datos debido a su falta de información sobre el tipo de datos y a su almacenamiento ineficiente.

El formato **CSV** (valores separados por comas) es uno de los formatos de archivo más populares que se utilizan para almacenar y transferir datos entre diferentes programas. Actualmente, muchas herramientas de gestión de bases de datos y el popular Excel ofrecen la importación y exportación de datos en este formato.

Cada línea del archivo representa un determinado conjunto de datos. Opcionalmente, en la primera línea podemos poner un encabezado que describa estos datos. Veamos un ejemplo sencillo de un archivo llamado `contacts.csv` que almacena los contactos de un teléfono:

```
Name,Phone
mother,222-555-101
father,222-555-102
wife,222-555-103
mother-in-law,222-555-104
```

En el archivo anterior, hay cuatro contactos que consisten en `nombre` y `número de teléfono`. Tenga en cuenta que la primera línea contiene un encabezado para ayudarlo a interpretar los datos.

Lectura de datos de un archivo CSV (parte 1)

La biblioteca estándar de Python ofrece un módulo llamado `csv` que proporciona funciones para leer y escribir datos en formato CSV. La lectura de datos se realiza utilizando el objeto `reader`, mientras que la escritura se realiza utilizando el objeto `writer`. Primero, analizaremos más de cerca la lectura de datos utilizando el objeto `reader`.

La función `reader` devuelve un objeto que le permite iterar sobre cada línea en el archivo CSV. Para crearlo, necesitamos pasar un objeto de archivo a la función `reader`. Para este propósito, podemos utilizar una función incorporada llamada open. Mire el código en el editor y ejecútelo.

Debería producir el siguiente resultado:

```
['Name', 'Phone']
['mother', '222-555-101']
['father', '222-555-102']
['wife', '222-555-103']
['mother-in-law', '222-555-104']
```

¿Qué sucedió? Hemos pasado un archivo abierto llamado `contacts.csv` y un separador utilizado para separar los datos del archivo a la función de lectura. El segundo argumento se puede omitir si nuestro archivo utiliza el separador predeterminado, que es una coma; lo hemos agregado para mostrarle cómo especificar otros separadores.

[1]:
```
import csv

with open('contacts.csv', newline='') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    for row in reader:
        print(row)
```

```
['Name', 'Phone']
['mother', '222-555-101']
['father', '222-555-102']
['wife', '222-555-103']
['mother-in-law', '222-555-104']
```

Lectura de datos de un archivo CSV (parte 2)

Por último, leemos cada fila mediante el bucle for. Observe que se devuelve una sola línea como una lista de cadenas. Sin embargo, se pueden obtener resultados más legibles, por ejemplo, mediante el método join. Observe el código en el editor y ejecútelo.

Debería producir el siguiente resultado:

```
Name,Phone
mother,222-555-101
father,222-555-102
wife,222-555-103
mother-in-law,222-555-104
```

NOTA: El argumento `newline=''` agregado a la función `open` nos protege de la interpretación incorrecta del carácter de nueva línea en diferentes plataformas.

[2]:
```
import csv

with open('contacts.csv', newline='') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    for row in reader:
        print(','.join(row))
```

```
Name,Phone
mother,222-555-101
father,222-555-102
wife,222-555-103
mother-in-law,222-555-104
```

Lectura de datos de un archivo CSV (parte 3)

El módulo `csv` proporciona una forma más cómoda de leer datos, en la que cada línea se asigna a un objeto `OrderedDict`. Para lograr esto, debemos utilizar la clase `DictReader` de la forma que mostramos en el editor.

El código en el editor producirá la siguiente salida:

```
mother : 222-555-101
father : 222-555-102
wife : 222-555-103
mother-in-law : 222-555-104
```

[3]:
```
import csv

with open('contacts.csv', newline='') as csvfile:
    reader = csv.DictReader(csvfile)
    for row in reader:
        print(row['Name'], ':', row['Phone'])
```

```
mother : 222-555-101
father : 222-555-102
wife : 222-555-103
mother-in-law : 222-555-104
```

Lectura de datos de un archivo CSV (parte 4)

Al igual que la función de lectura, la clase `DictReader` acepta un objeto `file` como argumento. Trata la primera línea del archivo como un encabezado desde el cual leer las claves. Si su archivo no tiene un encabezado, debe definirlo utilizando el argumento `fieldnames`. Mire el código en el editor.

NOTA: Si define más nombres de columnas que valores en el archivo, los valores faltantes serán `None`.

[4]:
```
import csv

with open('contacts.csv', newline='') as csvfile:
    fieldnames = ['Name', 'Phone']
```

```
    reader = csv.DictReader(csvfile, fieldnames=fieldnames)
    for row in reader:
        print(row['Name'], row['Phone'])
```

```
Name Phone
mother 222-555-101
father 222-555-102
wife 222-555-103
mother-in-law 222-555-104
```

Guardar datos en un archivo CSV (parte 1)

Como mencionamos antes, guardar datos en un archivo CSV se hace usando el objeto `writer` provisto por el módulo `csv`. Para crearlo, necesitamos usar una función llamada `writer`, que toma el mismo conjunto de argumentos que la función reader. Veamos cómo guardar contactos en un archivo CSV. Observa el código en el editor.

En el código de ejemplo, primero abrimos el archivo para escribir. El modo `'w'` crea un archivo para nosotros si aún no se ha creado. A continuación, creamos un objeto `writer` que usamos para agregar filas usando el método `writerow`. El método `writerow` toma una lista de valores como argumento y luego los guarda como una línea en un archivo CSV.

[5]:
```
import csv

with open('exported_contacts.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, delimiter=',')

    writer.writerow(['Name', 'Phone'])
    writer.writerow(['mother', '222-555-101'])
    writer.writerow(['father', '222-555-102'])
    writer.writerow(['wife', '222-555-103'])
    writer.writerow(['mother-in-law', '222-555-104'])
```

Guardar datos en un archivo CSV (parte 2)

Imagina una situación en la que agregas un contacto que contiene el separador utilizado para separar los valores en el archivo CSV. De manera predeterminada, estos valores están entre comillas, pero puedes cambiar esto con el argumento `quotechar`, que debe ser un solo carácter. Observa el código en el editor y observa cómo configuramos explícitamente los argumentos predeterminados.

El código guardará los siguientes datos en el archivo exported_contacts.csv:

```
Name,Phone
mother,222-555-101
father,222-555-102
wife,222-555-103
"grandmother, grandfather and auntie",222-555-105
```

El último argumento, llamado `quoting`, especifica qué valores deben `quotarse`. El valor predeterminado `QUOTE_MINIMAL` significa que solo se compondrán valores con caracteres especiales como separador o `quotechar`. En nuestro caso, es el valor de `"abuela, abuelo y tía"`.

A continuación, se muestran otras constantes que podemos usar como valor del argumento `quoting`:

- **csv.QUOTE_ALL**: comprime todos los valores
- **csv.QUOTE_NONNUMERIC**: comprime solo valores no numéricos
- **csv.QUOTE_NONE**: no comprime ningún valor. No es una buena idea establecer este valor si tiene caracteres especiales que requieren comillas, ya que esto generará un error.

NOTA: Los parámetros `quotechar` y `quoting` también se pueden usar en la función `reader`. Consulte la documentación para obtener más información.

[6]:
```
import csv

with open('exported_contacts.csv', 'w', newline='') as csvfile:
    writer = csv.writer(csvfile, delimiter=',', quotechar='"', quoting=csv.
 ↪QUOTE_MINIMAL)

    writer.writerow(['Name', 'Phone'])
    writer.writerow(['mother', '222-555-101'])
    writer.writerow(['father', '222-555-102'])
    writer.writerow(['wife', '222-555-103'])
    writer.writerow(['mother-in-law', '222-555-104'])
    writer.writerow(['grandmother, grandfather', '222-555-105'])
```

Guardar datos en un archivo CSV (parte 3)

¿Recuerdas cómo leíamos las filas del archivo CSV en objetos `OrderedDict`? En el módulo `csv`, hay una clase análoga llamada `DictWriter` con la que podemos mapear diccionarios a filas. A diferencia del objeto `DictReader`, al crear el objeto `DictWriter`, debemos definir un encabezado. Veamos el ejemplo en el editor.

Para crear el objeto `DictWriter`, usamos un objeto de archivo y una lista que contiene los nombres de las columnas. Ten en cuenta que antes de guardar el valor, primero llamamos al método `writeheader`, que agrega el encabezado a la primera línea del archivo. Después de eso, agregamos filas con valores pasando diccionarios al método `writerow`.

[7]:
```
import csv

with open('exported_contacts.csv', 'w', newline='') as csvfile:
    fieldnames = ['Name', 'Phone']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    writer.writerow({'Name': 'mother', 'Phone': '222-555-101'})
    writer.writerow({'Name': 'father', 'Phone': '222-555-102'})
    writer.writerow({'Name': 'wife', 'Phone': '222-555-103'})
    writer.writerow({'Name': 'mother-in-law', 'Phone': '222-555-104'})
    writer.writerow({'Name': 'grandmother, grandfather and auntie', 'Phone':␣
 ↪'222-555-105'})
```

2. **JSON (notación de objetos JavaScript)**

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

Atributos clave

JSON es un formato de intercambio de datos liviano, legible para humanos y fácil de entender tanto para humanos como para máquinas. Se usa ampliamente para representar datos estructurados, particularmente en aplicaciones web y API. JSON admite estructuras anidadas y se usa a menudo para datos semiestructurados. JSON se usa a menudo en desarrollo web, API y configuraciones donde los datos deben almacenarse o transmitirse en un formato estructurado. Es especialmente útil cuando se trabaja con datos semiestructurados.

Los archivos JSON almacenan datos como pares clave-valor en una estructura jerárquica. Se usan para tareas como almacenar configuraciones de aplicaciones, intercambiar datos entre servicios web y serializar estructuras de datos complejas en lenguajes de programación.

```
{
  "name": "Nnamdi Samuel",
  "age": 30,
  "email": "Nnamdi@example.com",
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "state": "CA",
    "zipcode": "12345"
  },
  "phone_numbers": [
    {
      "type": "home",
      "number": "555-1234"
    },
    {
      "type": "work",
      "number": "555-5678"
    }
  ]
}
```

¿Cómo podemos cumplir con este requisito?

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

JSON resuelve el problema utilizando 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 puede transportar datos mucho más complejos, recopilados y agregados en conjuntos más grandes.

Si desea transferir no solo datos sin procesar sino también todos los nombres vinculados 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?

Sí, podemos, pero no será JSON. Si desea que sus datos sean ampliamente comprendidos (no solo por sus contrapartes de Python), debe usar JSON.

Trabajar con el módulo JSON en Python

Ahora que estamos familiarizados con los aspectos esenciales de JSON, es hora de aprender a usarlo con Python. Nos preocupa un poco que pienses que queremos que construyas laboriosamente mensajes JSON, preocupándote por todos esos corchetes, paréntesis y dos puntos, y que descompongas líneas JSON complejas en factores primos. ¡Nada más lejos de la realidad! No tenemos la costumbre de pensar en ideas tan locas, aunque, para ser honestos, no es tan complejo como parece y estamos convencidos de que serías capaz de afrontar semejante desafío. Afortunadamente, no es necesario.

¿Por qué?

Porque hay un módulo de Python, llamado JSON, que puede realizar todas esas tareas pesadas por ti.

¿Cómo comenzamos una nueva aventura? Es obvio, y estamos seguros de que lo sabías antes de que te lo preguntáramos:

```
import json
```

La primera característica del módulo JSON es su capacidad de convertir automáticamente datos de Python (no todos y no siempre) en una cadena JSON. Si desea realizar dicha operación, puede utilizar una función llamada `dumps()`.

Nota: el 's al final del nombre de la función significa cadena. Existe una función muy similar con el nombre privado de este sufijo que escribe la cadena JSON en el archivo para transmisiones similares a archivos.

La función hace lo que promete: toma datos (incluso datos algo complicados) y produce una cadena llena con un mensaje JSON. Por supuesto, `dumps()` no es un profeta y no puede leer su mente, así que no espere milagros.

Comencemos con algunos fragmentos simples.

El primero de nuestros ejemplos toma un número y genera un número; no esperamos nada más:

```
import json
```

```
electron = 1.602176620898E10-19
print(json.dumps(electron))
```

El código genera:

```
16021766189.98
```

Nota: la notación es diferente, pero el valor sigue siendo el mismo. Compruébelo usted mismo.

Hagamos lo mismo, pero con una cadena, de esta manera:

```
import json

comics = '"The Meaning of Life" by Monty Python\'s Flying Circus'
print(json.dumps(comics))
```

El código genera:

```
"\"The Meaning of Life\" by Monty Python's Flying Circus"
```

Como puedes ver, se cumplieron todos los requisitos de JSON.

Ahora es un buen momento para presentar una lista. ¿Qué te parece este ejemplo?

```
import json

my_list = [1, 2.34, True, "False", None, ['a', 0]]
print(json.dumps(my_list))
```

Como puedes ver, en realidad hay dos listas. ¿Es eso un problema? ¡En absoluto!

El código imprime:

```
[1, 2.34, True, "False", null, ["a", 0]]
```

Queremos hacerte una pregunta: ¿qué sucederá si usamos una tupla en lugar de una lista? La respuesta es predecible: nada. Como JSON no puede distinguir entre listas y tuplas, ambas se convierten en matrices JSON.

Revisemos un diccionario. Aquí hay una prueba sencilla:

```
import json

my_dict = {'me': "Python", 'pi': 3.141592653589, 'data': (1, 2, 4, 8), 'set': None}
print(json.dumps(my_dict))
```

Y esta es la salida del código:

```
{"me": "Python", "pi": 3.141592653589, "data": [1, 2, 4, 8], "friend": "JSON", "set": null}
```

Ahora estamos listos para sacar algunas conclusiones.

Como puedes ver, Python usa 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

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?

Hay al menos dos opciones que podemos utilizar. La primera de ellas se basa en el hecho de que podemos sustituir la función que `dumps()` usa para obtener una representación textual de su argumento. Hay dos pasos a seguir:

- escribir tu propia función sabiendo cómo manejar tus objetos;
- hacer que `dumps()` lo sepa estableciendo el argumento de palabra clave llamado default;

Ahora mira el código en la ventana del editor. El ejemplo muestra una implementación simple de la idea.

El código imprime:

```
{"name": "John Doe", "age": 42}
```

Nota: decidimos usar el diccionario como destino del mensaje JSON. Gracias a eso, guardaremos los nombres de las propiedades junto con sus valores. Esto hará que JSON sea más fácil de leer y más comprensible para los humanos.

Nota: generar una excepción `TypeError` es obligatorio; esta es la única forma de informar a `dumps()` que su función no puede convertir objetos que no sean los derivados de la clase Who.

Nota: el proceso en el que un objeto (almacenado internamente por Python) se convierte en un aspecto textual o cualquier otro aspecto portátil a menudo se denomina serialización. De manera similar, la acción inversa (de portátil a interno) se denomina deserialización.

Como puede ver, hemos convertido (serializado) nuestro objeto en un diccionario; `dumps()` lo convertirá en un objeto JSON.

[8]:
```
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')


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

```
{"name": "John Doe", "age": 42}
```

El segundo 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`. Esto te da la oportunidad de sobrecargar el método que define una subclase de JSONEncoder y pasarlo a `dumps()` utilizando el argumento de palabra clave denominado `cls`, tal como en el código que hemos proporcionado en el editor.

Como puedes ver, estamos liberados de la obligación de generar excepciones. Genial, ¿no?

El código produce el mismo resultado que el anterior:

```
{"name": "John Doe", "age": 42}
```

Parece que sabemos lo suficiente sobre cómo viajar desde el mundo de Python al mundo de JSON, pero aún no sabemos cómo regresar. Vamos a ocuparnos de ello.

[9]:
```
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)


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

```
{"name": "John Doe", "age": 42}
```

La función que puede obtener una cadena JSON y convertirla en datos de Python se llama `loads()`: toma una cadena (de ahí la s al final de su nombre) e intenta crear una entidad de Python correspondiente a los datos recibidos.

Así es como funciona:

```
import json

jstr = '16021766189.98'
electron = json.loads(jstr)
print(type(electron))
print(electron)
```

El código imprime:

```
<class 'float'>
16021766189.98
```

La función `loads()` también puede manejar cadenas. Observa el fragmento:

```
import json

jstr = '"\\"The Meaning of Life\\" by Monty Python\'s Flying Circus"'
comics = json.loads(jstr)
print(type(comics))
print(comics)
```

¿Puedes ver las barras invertidas dobles dentro de jstr? ¿Son realmente necesarias?

Sí, lo son, ya que tenemos que entregar una cadena JSON exacta en `loads()`. Esto significa que la barra invertida debe preceder a todas las comillas existentes dentro de la cadena. Eliminar cualquiera de ellas hará que la cadena no sea válida y `loads()` no le gustará con seguridad.

El código genera lo siguiente:

```
<class 'str'>
```

```
"The Meaning of Life" by Monty Python's Flying Circus
```

¿Y qué pasa con las listas? ¿Es `loads()` lo suficientemente inteligente como para interpretarlas correctamente?

Sí, lo es. Eche un vistazo:

```
import json

jstr = '[1, 2.34, true, "False", null, ["a", 0]]'
my_list = json.loads(jstr)
print(type(mylist))
print(mylist)
```

El código imprime:

```
<class 'list'>
[1, 2.34, True, 'False', None, ['a', 0]]
```

Esperamos que el objeto JSON se procese correctamente.

Sí, lo hará:

```
import json

json_str = '{"me":"Python","pi":3.141592653589, "data":[1,2,4,8],"friend":"JSON","set": null}'
my_dict = json.loads(json_str)
print(type(my_dict))
print(my_dict)
```

El código imprime:

```
<class 'dict'>
{'me': 'Python', 'pi': 3.141592653589, 'data': [1, 2, 4, 8], 'friend': 'JSON', 'set': None}
```

Nuestras pruebas muestran que la tabla que presentamos antes funciona correctamente en ambas direcciones. Solo hay una diferencia específica: si un número codificado dentro de una cadena JSON no tiene ninguna parte fraccionaria, Python creará un número entero o, en caso contrario, un número flotante.

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.

[10]:
```
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 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.

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.

[11]:
```python
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}
```

0.1 Resumen:

[12]:
```python
import json

data = {
    "presidente": {
        "nombre": "Zaphod Beeblebrox",
        "especie": "Betelgeusian"
    }
}

# dumps: Serialización objeto python a str
json_string = json.dumps(data)
```

```
json_string
```

[12]:
```
'{"presidente": {"nombre": "Zaphod Beeblebrox", "especie": "Betelgeusian"}}'
```

[13]:
```
# loads: Deserialización string a objeto

json_decoded = json.loads(json_string)
json_decoded['presidente']
```

[13]:
```
{'nombre': 'Zaphod Beeblebrox', 'especie': 'Betelgeusian'}
```

[14]:
```
# Genera un .json con los datos de la variable data:

with open('archivo_json.json', 'w') as file:
     json.dump(data, file, indent=4)
```

[15]:
```
# Lectura del archivo .json:

with open('archivo_json.json', 'r') as file:
     print(json.load(file))
```

```
{'presidente': {'nombre': 'Zaphod Beeblebrox', 'especie': 'Betelgeusian'}}
```

Creado por:

Isabel Maniega