Lectura de archivos y URLs en pandas

Este documento resume los conceptos y ejemplos principales sobre cómo leer archivos locales y datos desde URLs usando pandas.

Lectura de archivos locales

Pandas permite leer distintos tipos de archivos para crear DataFrames:

- **CSV:**

```
import pandas as pd
df = pd.read_csv('ruta/al/archivo.csv')
```

- **JSON:**

```
df = pd.read_json('ruta/al/archivo.json')
```

- **Excel:**

```
df = pd.read_excel('ruta/al/archivo.xlsx')
```

- **XML:**

```
df = pd.read_xml('ruta/al/archivo.xml')
```

Lectura de datos desde URLs

También puedes leer archivos directamente desde una URL si el archivo está disponible públicamente:

A veces, para leer datos desde una URL que requiere autenticación o headers personalizados, puedes usar la librería `requests` para obtener el contenido y luego cargarlo en pandas:

```
import requests
import pandas as pd

url = 'https://ejemplo.com/archivo.csv'
response = requests.get(url)
df = pd.read_csv(pd.compat.StringIO(response.text))
```

Esto permite mayor flexibilidad al trabajar con APIs o fuentes protegidas.

- **CSV desde URL (usando requests):**

```
import requests
import pandas as pd

url =
'https://raw.githubusercontent.com/plotly/datasets/master/2014_usa_states.csv'
response = requests.get(url)
df = pd.read_csv(pd.compat.StringIO(response.text))
```

- **JSON desde URL (usando requests):**

```
import requests
import pandas as pd

url = 'https://jsonplaceholder.typicode.com/users'
response = requests.get(url)
df = pd.read_json(response.text)
```

- **Excel desde URL (usando requests):**

```
import requests
import pandas as pd

url = 'https://file-examples-
com.github.io/uploads/2017/02/file_example_XLS_10.xls'
response = requests.get(url)
with open('temp.xls', 'wb') as f:
    f.write(response.content)
df = pd.read_excel('temp.xls')
```

Lectura de JSON almacenado en una variable

Si tienes datos en formato JSON almacenados en una variable (por ejemplo, como un string o un diccionario en Python), puedes cargarlo directamente en pandas usando `pd.read_json` o `pd.json_normalize`:

- **JSON como string:**

```
import pandas as pd
import json

json_str = '[{"id": 1, "name": "Juan"}, {"id": 2, "name": "Ana"}]'
df = pd.read_json(json_str)
print(df)
```

- **JSON como diccionario (lista de dicts):**

```
import pandas as pd

data = [
    {"id": 1, "name": "Juan"},
    {"id": 2, "name": "Ana"}
]
df = pd.DataFrame(data)
print(df)
```

- **JSON anidado usando `json_normalize`:**

```
import pandas as pd

data = [
    {
        "id": 1,
        "name": "Juan",
        "address": {"city": "Santiago", "zip": "12345"}
    },
    {
        "id": 2,
        "name": "Ana",
        "address": {"city": "Valparaíso", "zip": "54321"}
    }
]
df = pd.json_normalize(data)
print(df)
```

Esto es útil cuando los datos provienen de una API, un archivo leído previamente, o se generan dinámicamente en tu código.

Explicación de `json_normalize`

La función `json_normalize` de pandas permite convertir datos en formato JSON anidado (estructuras con diccionarios y listas dentro de otros diccionarios) en un DataFrame plano, donde cada columna representa un campo del JSON.

Por ejemplo, si tienes un JSON como este:

```
{
    "id": 1,
    "name": "Juan",
    "address": {
        "city": "Santiago",
```

```
        "zip": "12345"
    }
}
```

Sin aplicar `json_normalize`, el contenido se vería en una tabla así:

```
| id | name |              address               |
|----|------|------------------------------------|
| 1  | Juan | {"city": "Santiago", "zip": "12345"}|
```

Y aplicándolo se verá así

```
| id | name | address.city | address.zip |
|----|------|--------------|-------------|
| 1  | Juan | Santiago     | 12345       |
| 2  | Ana  | Valparaíso   | 54321       |
```

Puedes usar `json_normalize` para obtener un DataFrame donde las claves anidadas se convierten en columnas:

```
import pandas as pd

data = [
        {
                "id": 1,
                "name": "Juan",
                "address": {"city": "Santiago", "zip": "12345"}
        },
        {
                "id": 2,
                "name": "Ana",
                "address": {"city": "Valparaíso", "zip": "54321"}
        }
]

df = pd.json_normalize(data)
print(df)
```

Antes de aplicar `json_normalize`, la tabla se vería así:

```
| id | name |               address                |
|----|------|--------------------------------------|
| 1  | Juan | {"city": "Santiago", "zip": "12345"}   |
| 2  | Ana  | {"city": "Valparaíso", "zip": "54321"} |
```

El resultado será:

```
| id | name | address.city | address.zip |
|----|------|--------------|-------------|
| 1  | Juan | Santiago     | 12345       |
| 2  | Ana  | Valparaíso   | 54321       |
```

Esto facilita el análisis de datos complejos provenientes de APIs o archivos JSON con estructuras anidadas.

Ejemplo: Lectura de datos desde una API pública que entrega múltiples archivos JSON

A continuación se muestra cómo obtener información de varios recursos de una API pública (por ejemplo, SWAPI), donde cada respuesta es un archivo JSON independiente. Los datos se recopilan en una lista y luego se normalizan en un DataFrame:

```python
import requests
import pandas as pd

personajes = []
for i in range(1, 11):
    url = f"https://swapi.dev/api/people/{i}/"
    response = requests.get(url)
    if response.status_code == 200:
        personajes.append(response.json())
    else:
        print(f"Error al obtener el personaje con ID {i}")

df = pd.json_normalize(personajes)
print(df[['name', 'height', 'mass', 'gender']])
```

Este ejemplo obtiene información de los primeros 10 personajes de Star Wars, almacena cada respuesta JSON en una lista y luego utiliza `json_normalize` para convertir los datos en un DataFrame plano.

Resultado:

name	height	mass	gender
Luke Skywalker	172	77	male
C-3PO	167	75	n/a
R2-D2	96	32	n/a
Darth Vader	202	136	male
Leia Organa	150	49	female
Owen Lars	178	120	male
Beru Whitesun lars	165	75	female
R5-D4	97	32	n/a

name	height	mass	gender
Biggs Darklighter	183	84	male
Obi-Wan Kenobi	182	77	male

Ejemplo: Aplicando `json_normalize` dos veces

A veces los datos JSON tienen varios niveles de anidación. Puedes aplicar `json_normalize` más de una vez para aplanar completamente la estructura.

Supongamos que tienes el siguiente JSON:

```
import pandas as pd

data = [
    {
        "id": 1,
        "name": "Juan",
        "address": {
            "city": "Santiago",
            "zip": "12345",
            "location": {"lat": -33.45, "lon": -70.66}
        }
    },
    {
        "id": 2,
        "name": "Ana",
        "address": {
            "city": "Valparaíso",
            "zip": "54321",
            "location": {"lat": -33.04, "lon": -71.62}
        }
    }
]
```

Primero, normalizas el nivel de `address`:

```
df1 = pd.json_normalize(data)
df1
```

Esto genera columnas como `address.city`, `address.zip`, y `address.location`.

> **Nota:** En versiones recientes de pandas, `json_normalize` puede aplanar automáticamente varios niveles de anidación en una sola llamada, sin necesidad de normalizar paso a paso. Esto simplifica el proceso y permite obtener todas las columnas anidadas directamente.

Luego, puedes normalizar el campo `address.location`:

```
df2 = pd.json_normalize(data, record_path=None, meta=[
    'id', 'name', ['address', 'city'], ['address', 'zip']
], record_prefix='', errors='ignore')

location_df = pd.json_normalize(df1['address.location'])
final_df = pd.concat([df1.drop(columns=['address.location']), location_df],
axis=1)
print(final_df)
```

Resultado final:

id	name	address.city	address.zip	lat	lon
1	Juan	Santiago	12345	-33.45	-70.66
2	Ana	Valparaíso	54321	-33.04	-71.62

Este método te permite aplanar estructuras JSON con múltiples niveles de anidación usando `json_normalize` varias veces.

Ejemplo: Normalizando una columna específica de un DataFrame

Si tienes un DataFrame donde una columna contiene datos en formato JSON anidado, puedes aplicar `json_normalize` solo a esa columna para aplanarla.

Supongamos que tienes el siguiente DataFrame:

```
import pandas as pd

data = [
    {"id": 1, "name": "Juan", "address": {"city": "Santiago", "zip": "12345"},
"location": {"lat": -33.45, "lon": -70.66}, "job": {"title": "Engineer",
"department": "IT"}},
    {"id": 2, "name": "Ana", "address": {"city": "Valparaíso", "zip": "54321"},
"location": {"lat": -33.04, "lon": -71.62}, "job": {"title": "Analyst",
"department": "Finance"}}
]
df = pd.DataFrame(data)
df
```

El resultado será:

Resultado:

id	name	address	location	job
1	Juan	{'city': 'Santiago', 'zip': '12345'}	{'lat': -33.45, 'lon': -70.66}	{'title': 'Engineer', 'department': 'IT'}

id	name	address	location	job
2	Ana	{'city': 'Valparaíso', 'zip': '54321'}	{'lat': -33.04, 'lon': -71.62}	{'title': 'Analyst', 'department': 'Finance'}

Para normalizar solo la columna `address` y unirla al DataFrame original:

```
address_df = pd.json_normalize(df['address'])
final_df = pd.concat([df.drop(columns=['address']), address_df], axis=1)
final_df
```

Resultado:

id	name	location	job	city	zip
1	Juan	{'lat': -33.45, 'lon': -70.66}	{'title': 'Engineer', 'department': 'IT'}	Santiago	12345
2	Ana	{'lat': -33.04, 'lon': -71.62}	{'title': 'Analyst', 'department': 'Finance'}	Valparaíso	54321

Este método es útil cuando solo una columna del DataFrame contiene datos anidados que necesitas aplanar.