



Published in Towards Data Science

You have **1** free member-only story left this month. [Sign up for Medium and get an extra one](#)



BChen

[Follow](#)Feb 22, 2021 · 8 min read · ✨ · [Listen](#)

Save



All Pandas `json_normalize()` you should know for flattening JSON

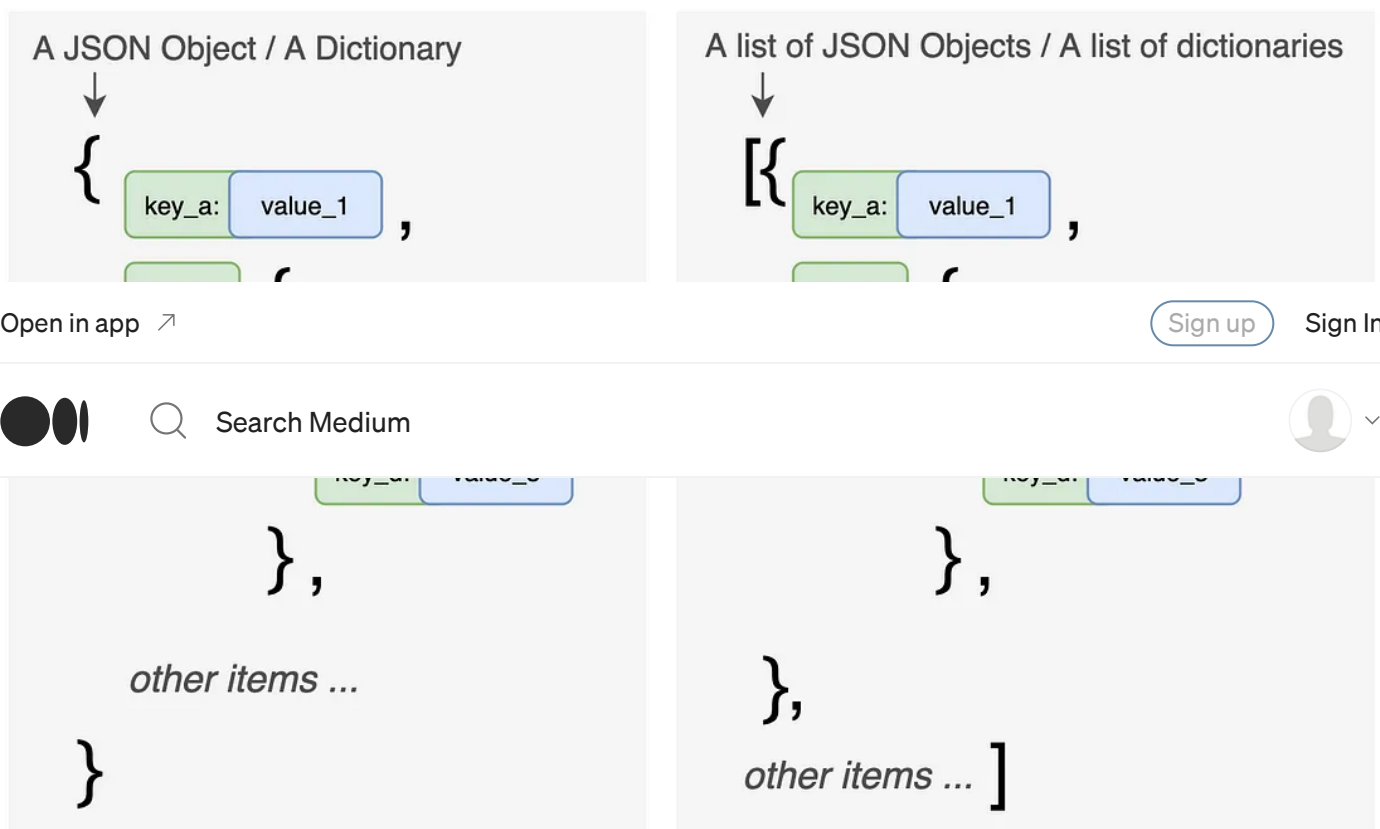
Some of the most useful Pandas tricks



All Pandas `json_normalize()` you should know for flattening JSON (Image by Author using [canva.com](#))

Reading data is the first step in any data science project. As a machine learning practitioner or a data scientist, you would have surely come across JSON (JavaScript Object Notation) data. JSON is a widely used format for storing and exchanging data. For example, NoSQL database like MongoDB store the data in JSON format, and REST API's responses are mostly available in JSON.

Although this format works well for storing and exchanging data, it needs to be converted into a tabular form for further analysis. You are likely to deal with 2 types of JSON structure, a JSON object or a list of JSON objects. In internal Python lingo, you are most likely to deal with a dict or a list of dicts.



A dictionary and a list of dictionaries (Image by author)

In this article, you'll learn how to use Pandas's built-in function `json_normalize()` to flatten those 2 types of JSON into Pandas DataFrames. This article is structured as follows:

1. Flattening a simple JSON
2. Flattening a JSON with multiple levels

3. Flattening a JSON with a nested list
4. Ignoring **KeyError** if keys are not always present
5. Custom separator using `sep`
6. Adding prefix for meta and record data
7. Working with a local file
8. Working with a URL

Please check out [Notebook](#) for the source code.

1. Flattening a simple JSON

Let's begin with 2 simple JSON, a simple dict and a list of simple dicts.



903



17

When the JSON is a simple dict

```
a_dict = {  
    'school': 'ABC primary school',  
    'location': 'London',  
    'ranking': 2,  
}
```

```
df = pd.json_normalize(a_dict)
```

	school	location	ranking
0	ABC primary school	London	2

(image by author)

The result looks great. Let's take a look at the data types with `df.info()`. We can see that columns that are numerical are cast to numeric types.

```
>>> df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1 entries, 0 to 0
Data columns (total 3 columns):
#   Column      Non-Null Count  Dtype
---  -
0   school      1 non-null      object
1   location    1 non-null      object
2   ranking     1 non-null      int64
dtypes: int64(1), object(2)
memory usage: 152.0+ bytes
```

When the data is a list of dicts

```
json_list = [
    { 'class': 'Year 1', 'student number': 20, 'room': 'Yellow' },
    { 'class': 'Year 2', 'student number': 25, 'room': 'Blue' },
]

pd.json_normalize(json_list)
```

	class	student number	room
0	Year 1	20	Yellow
1	Year 2	25	Blue

(image by author)

The result looks great. `json_normalize()` function is able to convert each record in the list into a row of tabular form.

What about keys that are not always present, for example, *num_of_students* is not available in the 2nd record.

```
json_list = [
    { 'class': 'Year 1', 'num_of_students': 20, 'room': 'Yellow' },
```

```
{ 'class': 'Year 2', 'room': 'Blue' }, # no num_of_students
]

pd.json_normalize(json_list)
```

	class	num_of_students	room
0	Year 1	20.0	Yellow
1	Year 2	NaN	Blue

(image by author)

We can see that no error is thrown and those missing keys are shown as NaN .

2. Flattening a JSON with multiple levels

Pandas `json_normalize()` works great for simple JSON (known as flattened JSON). What about JSON with multiple levels?

When the data is a dict

Let's first take a look at the following dict:

```
json_obj = {
    'school': 'ABC primary school',
    'location': 'London',
    'ranking': 2,
    'info': {
        'president': 'John Kasich',
        'contacts': {
            'email': {
                'admission': 'admission@abc.com',
                'general': 'info@abc.com'
            },
            'tel': '123456789',
        },
    },
}
```

The value of **info** is multiple levels (known as a nested dict). By calling `pd.json_normalize(json_obj)`, we get:

	school	location	ranking	info.president	info.contacts.email.admission	info.contacts.email.general	info.contacts.tel
0	ABC primary school	London	2	John Kasich	admission@abc.com	info@abc.com	123456789

The result looks great. All nested values are flattened and converted into separate columns.

If you don't want to dig all the way down to each value use the `max_level` argument. With the argument `max_level=1`, we can see that our nested value **contacts** is put up into a single column **info.contacts**.

```
pd.json_normalize(data, max_level=1)
```

	school	location	ranking	info.president	info.contacts
0	ABC primary school	London	2	John Kasich	{'email': {'admission': 'admission@abc.com', '...

(image by author)

When the data is a list of dicts

```
json_list = [
    {
        'class': 'Year 1',
        'student count': 20,
        'room': 'Yellow',
        'info': {
            'teachers': {
                'math': 'Rick Scott',
                'physics': 'Elon Mask'
            }
        }
    },
    {
        'class': 'Year 2',
        'student count': 25,
        'room': 'Blue',
        'info': {
```

```

    'teachers': {
        'math': 'Alan Turing',
        'physics': 'Albert Einstein'
    },
]

pd.json_normalize(json_list)

```

	class	student count	room	info.teachers.math	info.teachers.physics
0	Year 1	20	Yellow	Rick Scott	Elon Mask
1	Year 2	25	Blue	Alan Turing	Albert Einstein

(image by author)

We can see that all nested values in each record of the list are flattened and converted into separate columns. Similarly, we can use the `max_level` argument to limit the number of levels, for example

```
pd.json_normalize(json_list, max_level=1)
```

	class	student count	room	info.teachers
0	Year 1	20	Yellow	{'math': 'Rick Scott', 'physics': 'Elon Mask'}
1	Year 2	25	Blue	{'math': 'Alan Turing', 'physics': 'Albert Ein...'}

(image by author)

3. Flattening JSON with a nested list

What about JSON with a nested list?

When the data is a dict

Let's see how to flatten the following JSON into a DataFrame:

```

json_obj = {
    'school': 'ABC primary school',
    'location': 'London',
    'ranking': 2,
    'info': {
        'president': 'John Kasich',
        'contacts': {
            'email': {
                'admission': 'admission@abc.com',
                'general': 'info@abc.com'
            },
            'tel': '123456789',
        }
    },
    'students': [
        { 'name': 'Tom' },
        { 'name': 'James' },
        { 'name': 'Jacqueline' }
    ],
}

```

Notes the value of **students** is a nested list. By calling `pd.json_normalize(json_obj)`, we get:

	school	location	ranking	students	info.president	info.contacts.email.admission	info.contacts.email.general	info.contacts.tel
0	ABC primary school	London	2	[{'name': 'Tom'}, {'name': 'James'}, {'name': 'Jacqueline'}]	John Kasich	admission@abc.com	info@abc.com	123456789

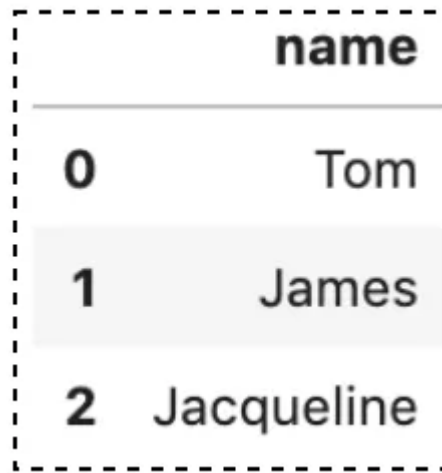
(image by author)

We can see that our nested list is put up into a single column **students** and other values are flattened. How can we flatten the nested list? To do that, we can set the argument `record_path` to `['students']`:

```

# Flatten students
pd.json_normalize(data, record_path=['students'])

```

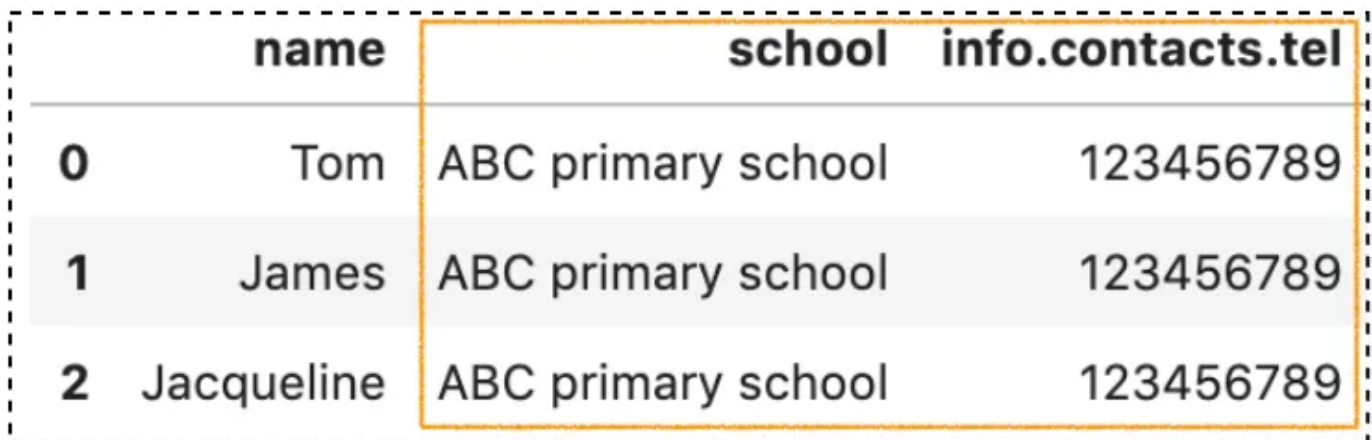



	name
0	Tom
1	James
2	Jacqueline

(image by author)

The result looks great but doesn't include **school** and **tel**. To include them, we can use the argument `meta` to specify a list of metadata we want in the result.

```
pd.json_normalize(
    json_obj,
    record_path=['students'],
    meta=['school', ['info', 'contacts', 'tel']],
)
```



	name	school	info.contacts.tel
0	Tom	ABC primary school	123456789
1	James	ABC primary school	123456789
2	Jacqueline	ABC primary school	123456789

(image by author)

When the data is a list of dicts

```
json_list = [
    {
```

```

'class': 'Year 1',
'student count': 20,
'room': 'Yellow',
'info': {
    'teachers': {
        'math': 'Rick Scott',
        'physics': 'Elon Mask'
    }
},
'students': [
    {
        'name': 'Tom',
        'sex': 'M',
        'grades': { 'math': 66, 'physics': 77 }
    },
    {
        'name': 'James',
        'sex': 'M',
        'grades': { 'math': 80, 'physics': 78 }
    }
],
},
{
    'class': 'Year 2',
    'student count': 25,
    'room': 'Blue',
    'info': {
        'teachers': {
            'math': 'Alan Turing',
            'physics': 'Albert Einstein'
        }
    },
    'students': [
        { 'name': 'Tony', 'sex': 'M' },
        { 'name': 'Jacqueline', 'sex': 'F' },
    ]
},
],
pd.json_normalize(json_list)

```

	class	student count	room	students	info.teachers.math	info.teachers.physics
0	Year 1	20	Yellow	[{'name': 'Tom', 'sex': 'M', 'grades': {'math': ...	Rick Scott	Elon Mask
1	Year 2	25	Blue	[{'name': 'Tony', 'sex': 'M'}, {'name': 'Jacqu...	Alan Turing	Albert Einstein

(image by author)

All nested lists are put up into a single column **students** and other values are flattened. To flatten the nested list, we can set the argument `record_path` to `['students']`. Notices that not all records have **math** and **physics**, and those missing values are shown as `NaN`.

```
pd.json_normalize(json_list, record_path=['students'])
```

	name	sex	grades.math	grades.physics
0	Tom	M	66.0	77.0
1	James	M	80.0	78.0
2	Tony	M	NaN	NaN
3	Jacqueline	F	NaN	NaN

(image by author)

If you would like to include other metadata use the argument `meta`:

```
pd.json_normalize(
    json_list,
    record_path = ['students'],
    meta=['class', 'room', ['info', 'teachers', 'math']]
)
```

	name	sex	grades.math	grades.physics	class	room	info.teachers.math
0	Tom	M	66.0	77.0	Year 1	Yellow	Rick Scott
1	James	M	80.0	78.0	Year 1	Yellow	Rick Scott
2	Tony	M	NaN	NaN	Year 2	Blue	Alan Turing
3	Jacqueline	F	NaN	NaN	Year 2	Blue	Alan Turing

(image by author)

4. The `errors` argument

The `errors` argument default to `'raise'` and will raise **KeyError** if keys listed in `meta` are not always present. For example, the math teacher is not available from the second record.

```
data = [
    {
        'class': 'Year 1',
        'student count': 20,
        'room': 'Yellow',
        'info': {
            'teachers': {
                'math': 'Rick Scott',
                'physics': 'Elon Mask',
            }
        },
        'students': [
            { 'name': 'Tom', 'sex': 'M' },
            { 'name': 'James', 'sex': 'M' },
        ]
    },
    {
        'class': 'Year 2',
        'student count': 25,
        'room': 'Blue',
        'info': {
            'teachers': {
                # no math teacher
                'physics': 'Albert Einstein'
            }
        },
        'students': [
            { 'name': 'Tony', 'sex': 'M' },
            { 'name': 'Jacqueline', 'sex': 'F' },
        ]
    },
]
```

A **KeyError** will be thrown when trying to flatten the math.

```
pd.json_normalize(  
    data,  
    record_path=['students'],  
    meta=['class', 'room', ['info', 'teachers', 'math']],  
)
```

```
-----  
KeyError                                Traceback (most recent call last)  
~/anaconda3/envs/tf-tutorial/lib/python3.7/site-packages/pandas/io/json/_normalize.py in _  
a, level)  
    323             try:  
--> 324                 meta_val = _pull_field(obj, val[level:])  
    325             except KeyError as e:  
  
~/anaconda3/envs/tf-tutorial/lib/python3.7/site-packages/pandas/io/json/_normalize.py in _  
    236         for field in spec:  
--> 237             result = result[field]  
    238         else:  
  
KeyError: 'math'
```

(image by author)

To work around it, set the argument `errors` to `'ignore'` and those missing values are filled with `NaN`.

```
pd.json_normalize(  
    data,  
    record_path=['students'],  
    meta=['class', 'room', ['info', 'teachers', 'math']],  
    errors='ignore'  
)
```

	name	sex	class	room	info.teachers.math
0	Tom	M	Year 1	Yellow	Rick Scott
1	James	M	Year 1	Yellow	Rick Scott
2	Tony	M	Year 2	Blue	NaN
3	Jacqueline	F	Year 2	Blue	NaN

(image by author)

5. Custom Separator using the `sep` argument

By default, all nested values will generate column names separated by `.`. For example `info.teachers.math`. To separate column names with something else, you can use the `sep` argument.

```
pd.json_normalize(
    data,
    record_path = ['students'],
    meta=['class', 'room', ['info', 'teachers', 'math']],
    sep='->'
)
```

	name	sex	grades->math	grades->physics	class	room	info->teachers->math
0	Tom	M	66.0	77.0	Year 1	Yellow	Rick Scott
1	James	M	80.0	78.0	Year 1	Yellow	Rick Scott
2	Tony	M	NaN	NaN	Year 2	Blue	Alan Turing
3	Jacqueline	F	NaN	NaN	Year 2	Blue	Alan Turing

(image by author)

6. Adding prefix for meta and record data

Sometimes, it may be more descriptive to add prefixes for the column names. To do that for the `meta` and `record_path`, we can simply pass the string to the argument `meta_prefix` and `record_prefix` respectively:

```
pd.json_normalize(
    data,
    record_path=['students'],
    meta=['class'],
    meta_prefix='meta-',
    record_prefix='student-'
)
```

	student-name	student-sex	student-grades.math	student-grades.physics	meta-class
0	Tom	M	66.0	77.0	Year 1
1	James	M	80.0	78.0	Year 1
2	Tony	M	NaN	NaN	Year 2
3	Jacqueline	F	NaN	NaN	Year 2

(image by author)

7. Working with a local file

Often, the JSON data you will be working on is stored locally as a `.json` file. However, Pandas `json_normalize()` function only accepts a dict or a list of dicts. To work around it, you need help from a 3rd module, for example, the Python `json` module:

```
import json
# load data using Python JSON module
with open('data/simple.json','r') as f:
    data = json.loads(f.read())

# Flattening JSON data
pd.json_normalize(data)
```

`data = json.loads(f.read())` loads data using Python `json` module. After that, `json_normalize()` is called on the data to flatten it into a DataFrame.

8. Working with a URL

JSON is a standard format for transferring data in REST APIs. Often, you need to work with API's response in JSON format. The simplest way to do that is using the Python `request` modules:

```
import requests
```

```
URL = 'http://raw.githubusercontent.com/BindiChen/machine-learning/master/data-analysis/027-pandas-convert-json/data/simple.json'
```

```
data = json.loads(requests.get(URL).text)
```

```
# Flattening JSON data  
pd.json_normalize(data)
```

Conclusion

Pandas `json_normalize()` function is a quick, convenient, and powerful way for flattening JSON into a DataFrame.

I hope this article will help you to save time in flattening JSON data. I recommend you to check out the [documentation](#) for the `json_normalize()` API and to know about other things you can do.

Thanks for reading. Please check out the [notebook](#) for the source code and stay tuned if you are interested in the practical aspect of machine learning.

You may be interested in some of my other Pandas articles:

- [Pandas cut\(\) function for transforming numerical data into categorical data](#)
- [Using Pandas method chaining to improve code readability](#)
- [How to do a Custom Sort on Pandas DataFrame](#)
- [All the Pandas shift\(\) you should know for data analysis](#)
- [When to use Pandas transform\(\) function](#)
- [Pandas concat\(\) tricks you should know](#)

- [Difference between `apply\(\)` and `transform\(\)` in Pandas](#)
- [All the Pandas `merge\(\)` you should know](#)
- [Working with datetime in Pandas DataFrame](#)
- [Pandas `read_csv\(\)` tricks you should know](#)
- [4 tricks you should know to parse date columns with Pandas `read_csv\(\)`](#)

More tutorials can be found on my [Github](#)

[Python](#)[Pandas](#)[Data Science](#)[Json](#)[Data Analysis](#)

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. [Take a look.](#)

By signing up, you will create a Medium account if you don't already have one. Review our [Privacy Policy](#) for more information about our privacy practices.



Get this newsletter

[About](#) [Help](#) [Terms](#) [Privacy](#)

Get the Medium app

