Loading, Converting, and Writing JSON Files

Print to PDF

Contents

Table of Contents

↑ Back to top

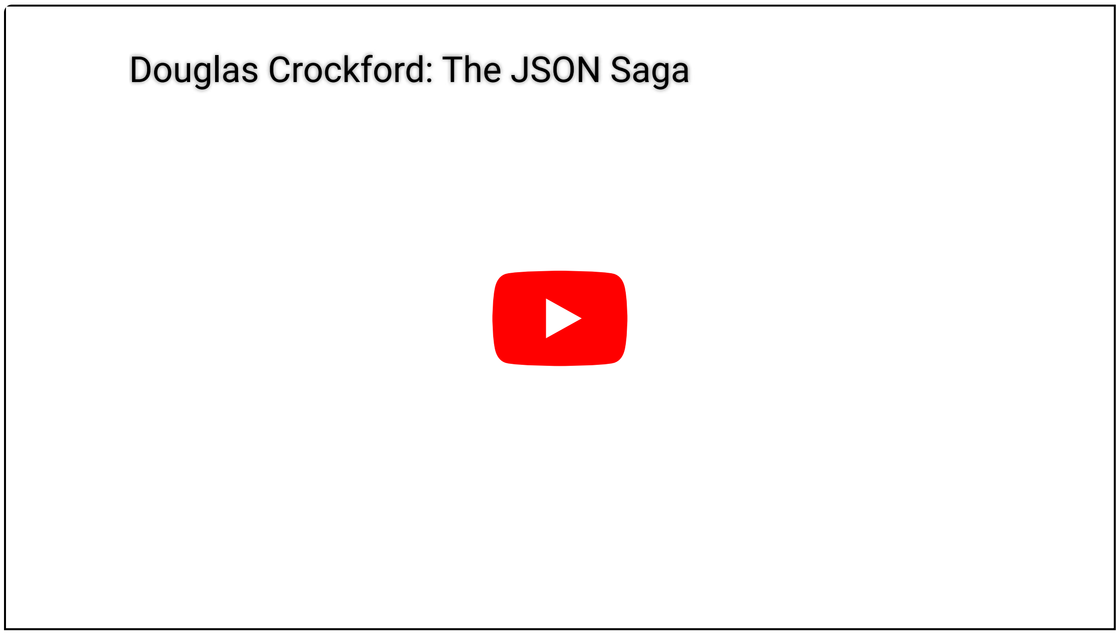
Skip to main content

3.1. Introduction: Douglas Crockford's JSON Saga

JSON stands for **JavaScript Object Notation**, and it is one of the most frequently used data formats for transferring data over the Internet. If you will be using a web-based data transfer system, such as an API (more on that in the next module), you will be dealing with data in JSON format.

The JSON data format is attributed to Douglas Crockford, a Javascript architect at Yahoo!, although in his words: "I discovered JSON. I do not claim to have invented JSON, because it already existed in nature. What I did was I found it, I named it, I described how it was useful." Here's an excellent talk by Douglas Crockford in which he describes the origin of JSON and the early struggles to create a universal and lightweight language for data transference. The talk is about 50 minutes long, but it is worth watching if you have the time.

```
from IPython.display import IFrame
IFrame(src="https://www.youtube.com/embed/-C-JoyNuQJs", width="560", height="315")
```



For what it's worth, Douglas Crockford calls the language "Jay-Sin", like the first name, and not "Jay-Sawn". Although he says "I strictly don't care" about however people choose to pronounce JSON.

Crockford and his collaborators developed JSON in 2001 with the objective of creating a language to store, display, and transfer in a way that works on every browser. In other words, the whole point of JSON is to be **universal**, so that the data are readable no matter the browser or environment. JSON is also designed to be as **lightweight** as possible: that is, JSON code is as compact as possible. JSON sacrificies features that expand the functionality of the language, such as comments, processing instructions, or attributes, in order to use the minimal amount of code. The lightweight construction of JSON makes JSON code much faster than alternative data interchange languages like XML. It also helps achieve the goal of universality: a lightweight language works within existing infrastructures with

Skip to main content

some form, and a minimalist data structure works with every language's idea of what data should be. In Crockford's words:

> One of the key design goals behind JSON was minimalism. My idea was that the less we have to agree on in order to inter-operate, the more likely we're going to be able to inter-operate well. If the interfaces are really simple, we can easily connect, and if the interfaces are really complicated, the likelihood that something's going to go wrong goes way, way up. So I endeavored to make JSON as simple as possible.

Before we get to examples of using JSONs in Python, load the following libraries:

```
import numpy as np
import pandas as pd
import json
import requests
import sys
sys.tracebacklimit = 0 # turn off the long tracebacks on error messages
#sys.tracebacklimit = None # use to turn tracebacks back on if needed
```

3.2. The Structure of a JSON File

The biggest difference between JSON (and other data interchange formats like XML, PHP, and YAML) and CSV is that JSON accomodates **tree-based** data structures, whereas CSV only handles **tabular** data. Many real-world applications require a tree-based structure for data. For example, here are the first two records from fake data from a business's customer records:

```
users = requests.get("https://jsonplaceholder.typicode.com/users")
print(users.text[0:1110])
```

```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
```

Skip to main content

```
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
      "name": "Romaguera-Crona",
      "catchPhrase": "Multi-layered client-server neural-net",
      "bs": "harness real-time e-markets"
    }
  },
  {
    "id": 2,
    "name": "Ervin Howell",
    "username": "Antonette",
    "email": "Shanna@melissa.tv",
    "address": {
      "street": "Victor Plains",
      "suite": "Suite 879",
      "city": "Wisokyburgh",
      "zipcode": "90566-7771",
      "geo": {
        "lat": "-43.9509",
        "lng": "-34.4618"
      }
    },
    "phone": "010-692-6593 x09125",
    "website": "anastasia.net",
    "company": {
      "name": "Deckow-Crist",
      "catchPhrase": "Proactive didactic contingency",
      "bs": "synergize scalable supply-chains"
    }
```

There are several elements of the JSON format that we need to discuss.

3.2.1. Lists, Sets, and Dictionaries

First, notice the opening square brace `[`. This character tells Python to read all the following records as elements of a **list**. In a Python list, the order of the elements matters and elements can be repeated, but are not given names. The first element is denoted with index 0, the second element is denoted with index 1, and so on. For example:

```
my_list = [5,8,-9]
my_list[0]
```

```
5
```

Within the JSON syntax, each individual element of the list is a **set**, denoted by curly braces `{` and `}`. A Python set differs from a Python list in that the order of the elements does not matter (it sorts the elements automatically), it doesn't allow repetition, and it allows the elements to be named. For

Skip to main content

example, in the following code, notice how the repeated 5s are removed and how the elements are sorted from smallest to largest:

```
my_set = {5,5,5,8,-9}
my_set
```

```
{-9, 5, 8}
```

It's only possible to use call individual elements of a set if those elements are given distinct names:

```
my_set = {'larry':5, 'curly':8, 'moe':-9}
my_set['larry']
```

```
5
```

In Python, sets in which elements of a set have names are called **dictionaries**. The names are called **keys**, and the elements themselves are called **values**. So in this case, an element of the dictionary has a key of "larry" and a value of 5. Like sets, dictionaries enforce no repetition, but they apply this restriction to the keys only: multiple keys can have the same value, but every value must have a distinct key. In this example, I define the key "larry" twice, and only the last "larry" entered into the dictionary definition is saved:

```
my_set = {'larry':15, 'larry':10, 'curly':10, 'moe':-9}
my_set
```

```
{'larry': 10, 'curly': 10, 'moe': -9}
```

That means that JSON notation is the same as a **list in which each element is a dictionary** in Python. Every dictionary represents one record in the data, and when we convert the data to a tabular dataframe, each record will occupy one row in the data.

3.2.2. Nested Structures

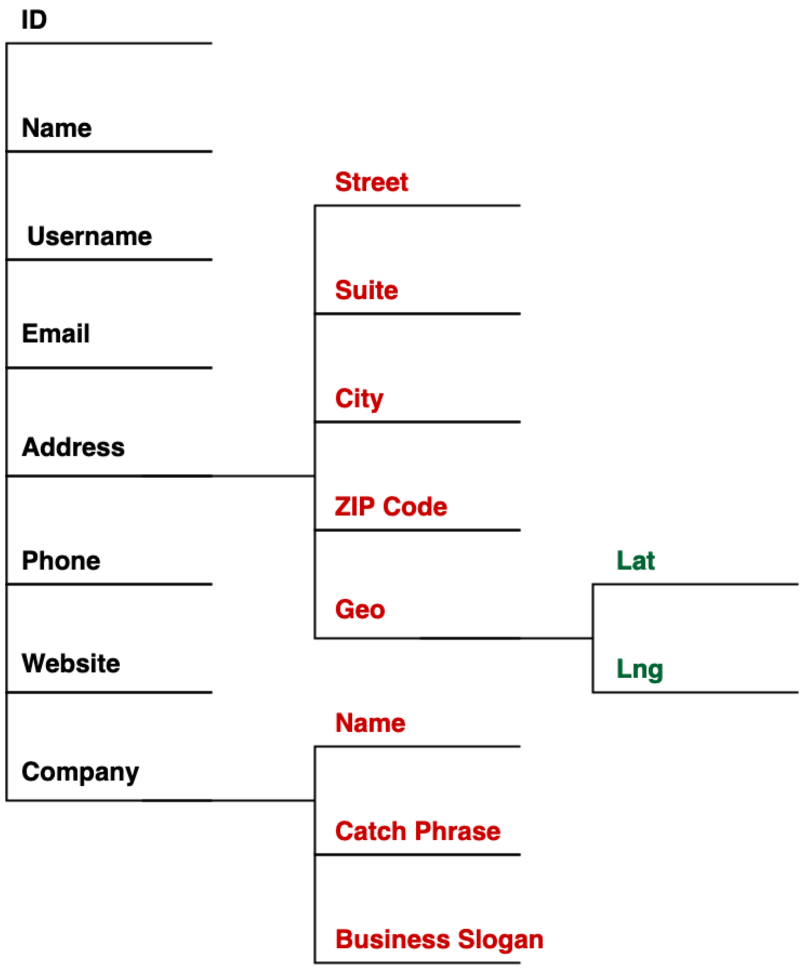
JSON can store data structures that are awkward or impossible for CSV. We can convert JSON formats to tabular formats to store the data in a data frame, but in doing so, we lose information about which features are nested within other fields.

Let's return to the example of the customer records stored in JSON format. In these data, some features are themselves dictionaries that contain additional features, resulting in a nested and tree-like shape to the data. The first record looks like:

```
print(users.text[0:560])
```

```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    },
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
      "name": "Romaguera-Crona",
      "catchPhrase": "Multi-layered client-server neural-net",
      "bs": "harness real-time e-markets"
    }
  },
```

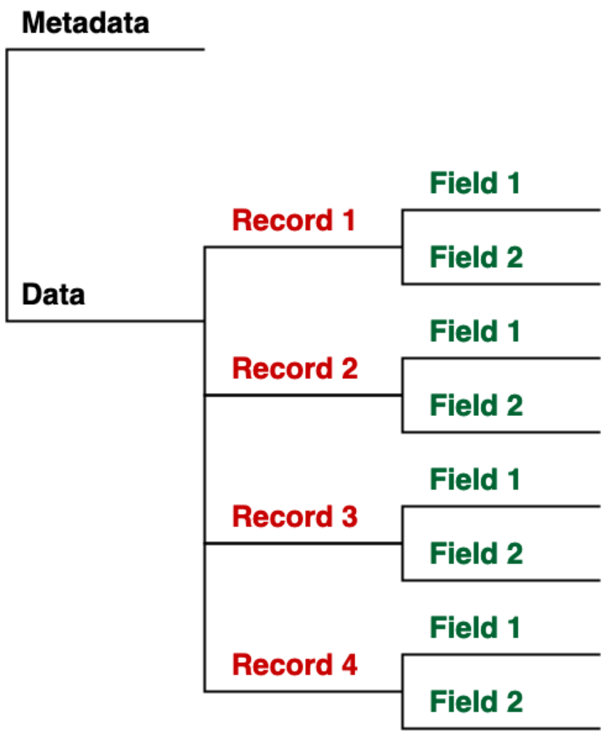
For this customer, we have the ID number, name, username, and email address. So far we can easily place these values in the same row of a table to store the same information. But the next key, `address`, is set equal to a dictionary, as indicated by the additional set of curly braces. Within the address field, we have data on the customer's street, suite, city, and ZIP code. We also have a field, `geo`, equal to yet another dictionary that contains the latitude and longitude coordinates of the address. The structure then returns to the first level of nesting, providing the phone number and website, before introducing another branch for company with three sub-fields: name, catch phrase, and business slogan. In all, the structure of this JSON data is best described with the following tree:



This tree can only be placed into a table, as we must do to work with a data frame, if we lose the information about nesting and instead treat the 15 distinct features as 15 columns in the data frame.

3.2.3. Metadata

In addition to nesting, JSON is an ideal format for placing metadata in the same file as the main data. Metadata is "data that provides information about other data". Metadata might describe the date the data were last accessed, provide the stable URL for accessing the data, might credit authorship or describe copyright, and so on, prior to displaying information about each of the records in the data. In general, JSON formats with metadata can look like this:



In order to convert JSON data with a metadata format to a tabular data frame, you will have to specify the name of the branch that contains the records. As with nested structures, this procedure involves losing information. In this case, we will lose the metadata.

3.2.4. Missing Values and Different Data Types

After unpacking the nested structure of a JSON data file and removing metadata, there can be differences from record to record in the information stored in the file. It's possible that some records do not have data on the same set of features that other records do. Even if the same feature is present across records, it is possible that its value has one data type in some records and another data type in other records.

For example, consider public opinion data in which records are individuals answering questions on a survey and the features include an individual ID number, the individual's rating of the Republican Candidate, the individual's rating of the Democratic candidate, and the individual's birthyear and gender. One possible way a few records can look like in JSON format (and then converted to a data frame) is:

```
case1 = '{"caseid":1.0,"ftrep":"Awful","ftdem":"Pretty good","birthyr":1960}'
case2 = '{"caseid":2.0,"ftrep":28.0,"ftdem":52.0,"birthyr":1987,"gender":2}'
case3 = '{"caseid":3.0,"ftrep":100.0,"ftdem":1.0,"gender":1}'
case_json = '[' + case1 + ',\n' + case2 + ',\n' + case3 + ']'
print(case_json)
```

Skip to main content

```
[{"caseid":1.0,"ftrep":"Awful","ftdem":"Pretty good","birthyr":1960},
{"caseid":2.0,"ftrep":28.0,"ftdem":52.0,"birthyr":1987,"gender":2},
{"caseid":3.0,"ftrep":100.0,"ftdem":1.0,"gender":1}]
```

```
pd.read_json(case_json)
```

	caseid	ftrep	ftdem	birthyr	gender
0	1	Awful	Pretty good	1960.0	NaN
1	2	28.0	52.0	1987.0	2.0
2	3	100.0	1.0	NaN	1.0

Take a moment to look at the raw JSON formatted data. There are some differences between the three cases. The first case did not record the individual's gender and the third case did not record the individual's birthyear. Note that in the JSON format, these features are simply absent: we do not need to explicitly code these values as missing. When we convert the data to tabular format, the cells for the missing features are filled with `NaN` missing values automatically.

Additionally, the first case records the feeling thermometer scores for the Republican and Democrat with strings: this person feels "Awful" towards the Republican and "Pretty good" towards the Democrat. For the second and third cases these ratings are coded numerically. The JSON format naturally allows the same feature to be populated by data of a different type across records. Tabular data, however, has stricter requirements for data uniformity within a column: specifically all of the data must have the same type in a column. The data frame handles that by coding all of these values with the ambiguous `object` type:

```
pd.read_json(case_json).dtypes
```

```
caseid       int64
ftrep       object
ftdem       object
birthyr    float64
gender     float64
dtype: object
```

In short, while it is straightforward to convert JSON data to tabular format, we can potentially lose a great deal of information in this conversion: nesting, metadata, and varying data type. In addition, tabular data inefficiently require that cells for missing values be filled with a symbol to denote missingness, to preserve the rectangular shape of the data frame.

Skip to main content

3.3. Loading and Reading JSON Data in Python

JSON, like CSV and ASCII, is a **text-based** system for data storage. When first loading JSON data into Python, the data will be read as a giant block of text. First we need to get Python to understand that the text is actually organized JSON data using the `json` library. Once we can work with the data as JSON, we can search through the JSON data's index path to extract particular datapoints. We can even construct loops to pull out lists of values from across the records.

3.3.1. Using the `requests.get()`, `json.loads()` and `json.dumps()` Functions

There is an important function from the `requests` library and two important functions in the `json` library for us to know when working with JSON data:

- `requests.get()` downloads data from a URL and stores it in Python's memory as a giant block of text. It also can include additional parameters to send to the website to provide information on credentials or on specifying the subset of data to collect.
- `json.loads()` converts text into an object that Python recognizes as JSON-formatted data.
- `json.dumps()` converts JSON data into a block of text.

Consider again the JSON data on fake consumer records. The data are stored on a website called JSON Placeholder, which has several excellent example JSON datasets and has resources for guiding people who are writing their own APIs. The customer data exists here: https://jsonplaceholder.typicode.com/users. Take a moment and click on this link to see how the data appear in your web browser.

To download the data, use `requests.get()`:

```
users = requests.get("https://jsonplaceholder.typicode.com/users")
```

To access the text of this download, use the `.text` attribute. If I want to display only the first record, I have to specify the first and last character numbers of this record, which turns out to be (after a lot of guess-and-checking) the first 560 characters:

```
print(users.text[0:560])
```

```
[
  {
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    },
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
      "name": "Romaguera-Crona",
      "catchPhrase": "Multi-layered client-server neural-net",
      "bs": "harness real-time e-markets"
    }
  },
```

Presently, although the text above looks like it is formatted like a JSON file, Python only understands it as text (with type `str`):

```
type(users.text)
```

```
str
```

To get Python to read this text as JSON data, make sure the `json` library is imported (which we did at the top of this notebook), and use the `json.loads()` function. Now, to display the first record, I only have to pass the index 0, representing the first item in a list:

```
users_json = json.loads(users.text)
users_json[0]
```

```
{'id': 1,
 'name': 'Leanne Graham',
 'username': 'Bret',
 'email': 'Sincere@april.biz',
 'address': {'street': 'Kulas Light',
  'suite': 'Apt. 556',
  'city': 'Gwenborough',
  'zipcode': '92998-3874',
  'geo': {'lat': '-37.3159', 'lng': '81.1496'}},
 'phone': '1-770-736-8031 x56442',
 'website': 'hildegard.org',
 'company': {'name': 'Romaguera-Crona',
  'catchPhrase': 'Multi-layered client-server neural-net',
  'bs': 'harness real-time e-markets'}}
```

Instead of text, Python now recognizes `users_json` as a list:

```
type(users_json)
```

```
list
```

3.3.2. Searching Along the JSON Index Path

After using `json.loads()`, Python understands the JSON data to be a list-of-lists. We can now use indices to extract particular datapoints. The records are numbers, starting with 0, and keys within particular records can be called by name.

For example, to extract the email address of the fifth customer (element 4), in the data, I type:

```
users_json[4]['email']
```

```
'Lucio_Hettinger@annie.ca'
```

If we call fields that contain several nested features, Python returns a dictionary:

```
users_json[0]['address']
```

```
{'street': 'Kulas Light',
 'suite': 'Apt. 556',
 'city': 'Gwenborough',
 'zipcode': '92998-3874',
 'geo': {'lat': '-37.3159', 'lng': '81.1496'}}
```

To extract elements that are several levels of nesting down the JSON tree, specify the keys in order that lead to the desired element. For example, to extract the latitudinal coordinate, we navigate to the `address`, `geo`, and `lat` keys:

```
users_json[0]['address']['geo']['lat']
```

```
'-37.3159'
```

Finally, it may be necessary at times to covert the JSON back to text, maybe to assist in transfering the data to another platform (although I recommend using the `.to_json()` method, discussed below, for this purpose). To convert JSON back to text, use the `json.dumps()` function:

```
users_text = json.dumps(users_json)
type(users_text)
```

```
str
```

3.3.3. Looping Across Records to Extract Datapoints

While JSON is a well-organized and flexible data format that preserves a lot of the meaningful context of the data, most analytical tools and statistical models can only operate on tabular data. So a major challenge with JSON data is converting the data to a tabular format to be saved in a data frame. One method is to construct a loop across records to extract desired elements.

Before we discuss this method, I do not recommend using this approach in general. Loops are usually slow, and it is much faster to perform these kinds of operations by **vectorizing** code. Vectorizing refers to the ability of a programming language to operate on an entire vector of data and work on several elements simultaneously, rather than on each element one at a time. Vectorization is the specialty of the `numpy` library, and most functions in `pandas` are vectorized as well. I suggest using `pd.read_json()` and `pd.json_normalize()` to convert JSON data to a data frame as they will be several orders of magnitude faster for converting the entire JSON to a data frame. Looping make sense only when we need to extract a small number of features from JSON data with a large feature set, because with the loop we directly call the features we want and we can avoid having to work with features we don't need.

To start the loop, choose an index to represent one record. To display all of the email addresses, type:

```
for u in users_json:
    print(u['email'])
```

```
Sincere@april.biz
Shanna@melissa.tv
Nathan@yesenia.net
Julianne.OConner@kory.org
```

Skip to main content

```
Telly.Hoeger@billy.biz
Sherwood@rosamond.me
Chaim_McDermott@dana.io
Rey.Padberg@karina.biz
```

Alternatively, to place all of the email addresses in a list, you can loop inside of a list like this:

```
emails = [u['email'] for u in users_json]
emails
```

```
['Sincere@april.biz',
 'Shanna@melissa.tv',
 'Nathan@yesenia.net',
 'Julianne.OConner@kory.org',
 'Lucio_Hettinger@annie.ca',
 'Karley_Dach@jasper.info',
 'Telly.Hoeger@billy.biz',
 'Sherwood@rosamond.me',
 'Chaim_McDermott@dana.io',
 'Rey.Padberg@karina.biz']
```

The trick is extracting the data and saving it in a data frame at the same time. We can do this by using the `pd.DataFrame()` function, and using list loops inside this function:

```
users_df = pd.DataFrame(
    [u['name'], u['email'], u['company']['name']] for u in users_json
)
users_df.columns = ['name', 'email', 'company_name']
users_df
```

Skip to main content

	name	email	company_name
0	Leanne Graham	Sincere@april.biz	Romaguera-Crona
1	Ervin Howell	Shanna@melissa.tv	Deckow-Crist
2	Clementine Bauch	Nathan@yesenia.net	Romaguera-Jacobson
3	Patricia Lebsack	Julianne.OConner@kory.org	Robel-Corkery
4	Chelsey Dietrich	Lucio_Hettinger@annie.ca	Keebler LLC
5	Mrs. Dennis Schulist	Karley_Dach@jasper.info	Considine-Lockman
6	Kurtis Weissnat	Telly.Hoeger@billy.biz	Johns Group
7	Nicholas Runolfsdottir V	Sherwood@rosamond.me	Abernathy Group
8	Glenna Reichert	Chaim_McDermott@dana.io	Yost and Sons
9	Clementina DuBuque	Rey.Padberg@karina.biz	Hoeger LLC

Please, note, there are many many ways to construct a loop to extract JSON elements into a dataframe. If you look on Stack Overflow, for example, you will see many different approaches, and it can be confusing. Find an approach that you understand and feel comfortable using, and go with that. There's not much difference between one loop and the next, as the real improvement comes from vectorization.

3.4. Using `pd.read_json()` and `pd.json_normalize()` to Store JSON Data in a Data Frame

If there aren't too many features in the data, or if you want to keep all of the features in the data anyway, then the fastest and best way to convert JSON data to a tabular data frame depends on **whether or not the JSON data has metadata or a nested structure**.

3.4.1. Situation 1: No nesting, no metadata

If the JSON has no nesting - that is, if every key is associated with a single datapoint, and no key is associated with a dictionary that contains additional features - and also does not include metadata, then use the following steps:

1. Use `requests.get()` to download the raw JSON data (unless you have another way of acquiring

2. Use `pd.read_json()` on the `.text` attribute of the output of `requests.get()`

The result will be a dataframe in which the columns have the same names as the keys in the JSON file. For example, JSON Placeholder has an example JSON dataset that contains random Latin posts to a blog. This JSON contains no metadata and no nesting. The best way to acquire this dataset and convert it to a dataframe is as follows:

```
posts = requests.get("https://jsonplaceholder.typicode.com/posts")
posts_df = pd.read_json(posts.text)
posts_df
```

	userId	id	title	body
0	1	1	sunt aut facere repellat provident occaecati e...	quia et suscipit\nsuscipit recusandae consequu...
1	1	2	qui est esse	est rerum tempore vitae\nsequi sint nihil repr...
2	1	3	ea molestias quasi exercitationem repellat qui...	et iusto sed quo iure\nvoluptatem occaecati om...
3	1	4	eum et est occaecati	ullam et saepe reiciendis voluptatem adipisci\...
4	1	5	nesciunt quas odio	repudiandae veniam quaerat sunt sed\nalias aut...
...	...	...	...	...
95	10	96	quaerat velit veniam amet cupiditate aut numqu...	in non odio excepturi sint eum\nlabore volupta...
96	10	97	quas fugiat ut perspiciatis vero provident	eum non blanditiis soluta porro quibusdam volu...
97	10	98	laboriosam dolor voluptates	doloremque ex facilis sit sint culpa\nsoluta a...
98	10	99	temporibus sit alias delectus eligendi possimu...	quo deleniti praesentium dicta non quod\naut e...
99	10	100	at nam consequatur ea labore ea harum	cupiditate quo est a modi nesciunt soluta\nips...

100 rows × 4 columns

3.4.2. Situation 2: Nesting, but no metadata

If the JSON file contains nesting, but no metadata, then the best strategy is to

1. Use `requests.get()` to download the raw JSON data (unless you have another way of acquiring the raw data)
2. Use `json.loads()` on the `.text` attribute of the output from step 1 to register the data as a list in Python
3. Use the `pd.json_normalize()` function on the list that is the output of step 2

The `pd.json_normalize()` function stores every feature in the data in a separate column, no matter how many levels of nesting it must parse to find the feature.

Every column has the same name as the key from which it drew the feature. For features that are nested within other features, `pd.json_normalize()` uses every key on the path to the datapoint to construct the column name, separated by periods. For example, the `users` data that we worked with above contains up to three levels of nesting. So the `lat` data is stored in a column named `address.geo.lat`, since we had to navigate to “address”, then “geo”, then “lat” in the JSON to find these data:

```
users = requests.get("https://jsonplaceholder.typicode.com/users")
users_json = json.loads(users.text)
users_df = pd.json_normalize(users_json)
users_df
```

	id	name	username	email	phone	websit
0	1	Leanne Graham	Bret	Sincere@april.biz	1-770-736-8031 x56442	hildegard.or
1	2	Ervin Howell	Antonette	Shanna@melissa.tv	010-692-6593 x09125	anastasia.ne
2	3	Clementine Bauch	Samantha	Nathan@yesenia.net	1-463-123-4447	ramiro.inf
3	4	Patricia Lebsack	Karianne	Julianne.OConner@kory.org	493-170-9623 x156	kale.b
4	5	Chelsey Dietrich	Kamren	Lucio_Hettinger@annie.ca	(254)954-1289	demarco.inf
5	6	Mrs. Dennis Schulist	Leopoldo_Corkery	Karley_Dach@jasper.info	1-477-935-8478 x6430	ola.or
6	7	Kurtis Weissnat	Elwyn.Skiles	Telly.Hoeger@billy.biz	210.067.6132	elvis.i
7	8	Nicholas Runolfsdottir V	Maxime_Nienow	Sherwood@rosamond.me	586.493.6943 x140	jacynthe.cor
8	9	Glenna Reichert	Delphine	Chaim_McDermott@dana.io	(775)976-6794 x41206	conrad.cor
9	10	Clementina DuBuque	Moriah.Stanton	Rey.Padberg@karina.biz	024-648-3804	ambrose.ne

3.4.3. Situation 3: Metadata

Finally, if the JSON file contains metadata (regardless of whether or not the records contain nested data), follow these steps:

Skip to main content

1. Use `requests.get()` to download the raw JSON data (unless you have another way of acquiring the raw data)
2. Use `json.loads()` on the `.text` attribute of the output from step 1 to register the data as a list in Python
3. Look at the data using a web-browser, or using the text that appears when the JSON object is called in Python, to decide on the path that is necessary to find the records
4. Use the `pd.json_normalize()` function on the list that is the output of step 2. But within this function, use the `record_path` parameter and set it equal to a list with the keys, in order, that lead to the record
5. Optionally, use the `meta` and `meta_prefix` parameters in `pd.json_normalize()` to store the metadata in the dataframe

For example, we can access a JSON file for the top 25 top posts at the moment on Reddit's r/popular page here: http://www.reddit.com/r/popular/top.json. If our goal is to construct a data frame with 25 rows, one for each post, we must find the path that leads to these data. Looking at the web-browser, the top-level has two keys: "kind" and "data". "kind" is simply metadata telling us that this file contains listings, and the data live in "data". Within this branch, there are four more metadata branches, "modhash", "dist", "before", and "after", and the data we need exist within "children". So the path we need is `["data", "children"]`. The code to construct the data frame we need is:

```python
url = "http://www.reddit.com/r/popular/top.json"
reddit = requests.get(url, headers = {'User-agent': 'DS6001'})
reddit_json = json.loads(reddit.text)
reddit_df = pd.json_normalize(reddit_json, record_path = ["data", "children"])
reddit_df
```

	kind	data.approved_at_utc	data.subreddit	data.selftext	data.author_fullname
0	t3	None	facepalm		t2_rtjjhl9i
1	t3	None	technicallythetruth		t2_mqxf3ma8a
2	t3	None	Damnthatsinteresting		t2_tbyapini
3	t3	None	me_irl		t2_5eq0rhis
4	t3	None	pics		t2_afqso
5	t3	None	meirl		t2_9dckfrpcw
6	t3	None	TikTokCringe	I just found this old tiktok buried on my SD card	t2_a5qeyx6
7	t3	None	cursedcomments		t2_tm29kwsn
8	t3	None	ImTheMainCharacter		t2_55zvfn1v
9	t3	None	perfectlycutscreams		t2_gqms9jrk
10	t3	None	Funnymemes		t2_uece04xp
11	t3	None	meirl		t2_xn3vj

	kind	data.approved_at_utc	data.subreddit	data.selftext	data.author_fullname
13	t3	None	me_irl		t2_be9vtcbr
14	t3	None	meirl		t2_7s141vzd
15	t3	None	mildlyinfuriating		t2_17ake5
16	t3	None	mildlyinfuriating		t2_tm26fszp
17	t3	None	MadeMeSmile		t2_ki3i2
18	t3	None	MadeMeSmile		t2_l9kr8ic8
19	t3	None	me_irl		t2_jnjt6px
20	t3	None	coolguides		t2_10m3r2
21	t3	None	memes		t2_8q3doyn4
22	t3	None	oddlysatisfying		t2_aanbbl0
23	t3	None	wholesomememes		t2_ad7p5vnzk
24	t3	None	nextfuckinglevel		t2_58wucxff4

	kind	data.approved_at_utc	data.subreddit	data.selftext	data.author_fullname

25 rows × 149 columns

Please note: we added `headers = {'User-agent': 'DS6001'}` to the `requests.get()` function. When we use `requests.get()`, we are accessing an API: a web-based interface for transferring data, usually in the form of JSON files. We will cover APIs in the next module. Most APIs, including Reddit's, include some security to keep any one user from overusing the API and causing it to crash. For Reddit, we only need to provide a unique "User-agent", which is an ID that Reddit uses to keep track of how much we are using the API. If two people use the same User-agent, both uses are counted against the same limit, so it's better to choose a User-agent that has a unique name. If this block of code results in an error for you (so that `reddit.text` displays as `'{"message": "Too Many Requests", "error": 429}'`), then change `DS6001` to something else, and it should work.

To save the metadata in the data frame, use the `meta` parameter, set equal to a list of paths for each metadata feature you wish to store in the dataframe. In this case, to save the "kind" feature and the "after" feature within the "data" branch, I type `meta = ['kind', ['data', 'after']]`. Finally, because there is already a `kind` feature stored within the records, I must distinguish the metadata feature with a prefix. I use `meta_prefix='meta'` to place "meta" prior to the column names of the metadata.

Because metadata exist outside of the records, they will be constant across the rows in the dataframe. Here's the code to convert the Reddit data to a data frame while storing the metadata. Take a look at the resulting dataframe and scroll all the way to the right to see the two metadata columns:

```
url = "http://www.reddit.com/r/popular/top.json"
reddit = requests.get(url, headers = {'User-agent': 'DS6001'})
reddit_json = json.loads(reddit.text)
reddit_df = pd.json_normalize(reddit_json,
                              record_path = ["data", "children"],
                              meta = ['kind', ['data', 'after']],
                              meta_prefix = "meta")
reddit_df
```

Skip to main content

	kind	data.approved_at_utc	data.subreddit	data.selftext	data.author_fullname
0	t3	None	facepalm		t2_rtjjhl9i
1	t3	None	technicallythetruth		t2_mqxf3ma8a
2	t3	None	Damnthatsinteresting		t2_tbyapini
3	t3	None	me_irl		t2_5eq0rhis
4	t3	None	pics		t2_afqso
5	t3	None	meirl		t2_9dckfrpcw
6	t3	None	TikTokCringe	I just found this old tiktok buried on my SD card	t2_a5qeyx6
7	t3	None	cursedcomments		t2_tm29kwsn
8	t3	None	ImTheMainCharacter		t2_55zvfn1v
9	t3	None	perfectlycutscreams		t2_gqms9jrk
10	t3	None	Funnymemes		t2_uece04xp
11	t3	None	meirl		t2_xn3vj

Skip to main content

	kind	data.approved_at_utc	data.subreddit	data.selftext	data.author_fullname
13	t3	None	me_irl		t2_be9vtcbr
14	t3	None	meirl		t2_7s141vzd
15	t3	None	mildlyinfuriating		t2_17ake5
16	t3	None	mildlyinfuriating		t2_tm26fszp
17	t3	None	MadeMeSmile		t2_ki3i2
18	t3	None	MadeMeSmile		t2_l9kr8ic8
19	t3	None	me_irl		t2_jnjt6px
20	t3	None	coolguides		t2_10m3r2
21	t3	None	memes		t2_8q3doyn4
22	t3	None	oddlysatisfying		t2_aanbbl0
23	t3	None	wholesomememes		t2_ad7p5vnzk
24	t3	None	nextfuckinglevel		t2_58wucxff4

Skip to main content

kind	data.approved_at_utc	data.subreddit	data.selftext	data.author_fullname

25 rows × 151 columns

3.5. Saving JSON Files and Converting Data Frames to JSON

It is possible to save a JSON file to your local disk space, or to convert a dataframe to JSON and, if you want to, save that JSON to disk.

3.5.1. Saving Existing JSON Files to Disk

Once we've used the `json.loads()` function to register data as JSON in Python, we can save that JSON to our local disk space by using `open()` to create a new file, and `json.dump()` (note: not `json.dumps()`) to save the JSON to that file.

For example, we registered the customer data as JSON in Python with the following code:

```
users = requests.get("https://jsonplaceholder.typicode.com/users")
users_json = json.loads(users.text)
```

To save this JSON file to my harddrive, I first use `os.chdir()` to set the working directory to the folder where I want to save my JSON file. (I omit that code here because it won't work on other people's computers.) Then I type:

```
with open('users.json', 'w') as outfile:
     json.dump(users_json, outfile, sort_keys = True, indent = 4,
               ensure_ascii = False)
```

It is also possible to use a file extension such as `.txt` instead of `.json` to save the JSON formatted data in a plain text file.

Skip to main content

3.5.2. Converting Tabular DataFrames to JSON

Any dataframe can be turned into a JSON file by applying the `.to_json()` method to the dataframe that is saved in Python's memory. The trick is specifying a good organization for this file. Recall that JSON structures are much more flexible than dataframes. It is not possible to go from a dataframe to a nested, complicated JSON structure because the information about nesting simply does not exist for dataframes. That said, there are several choices for organizing the data:

- `orient="records"` works with JSON files organized as a list-of-sets, where each set is an entire record (or a row in flat data):

```
users_df = users_df.loc[0:2,] # just keeping the first 3 records, for display purposes
new_json = users_df.to_json(orient="records")
json.loads(new_json)[0]
```

```
{'id': 1,
 'name': 'Leanne Graham',
 'username': 'Bret',
 'email': 'Sincere@april.biz',
 'phone': '1-770-736-8031 x56442',
 'website': 'hildegard.org',
 'address.street': 'Kulas Light',
 'address.suite': 'Apt. 556',
 'address.city': 'Gwenborough',
 'address.zipcode': '92998-3874',
 'address.geo.lat': '-37.3159',
 'address.geo.lng': '81.1496',
 'company.name': 'Romaguera-Crona',
 'company.catchPhrase': 'Multi-layered client-server neural-net',
 'company.bs': 'harness real-time e-markets'}
```

- `orient="columns"` works with JSON files organized as a list-of-dictionaries, where each dictionary is an entire column (the names are the row-names in the tabular data)

```
new_json = users_df.to_json(orient="columns")
json.loads(new_json)
```

```
{'id': {'0': 1, '1': 2, '2': 3},
 'name': {'0': 'Leanne Graham', '1': 'Ervin Howell', '2': 'Clementine Bauch'},
 'username': {'0': 'Bret', '1': 'Antonette', '2': 'Samantha'},
 'email': {'0': 'Sincere@april.biz',
  '1': 'Shanna@melissa.tv',
  '2': 'Nathan@yesenia.net'},
 'phone': {'0': '1-770-736-8031 x56442',
  '1': '010-692-6593 x09125',
  '2': '1-463-123-4447'},
 'website': {'0': 'hildegard.org', '1': 'anastasia.net', '2': 'ramiro.info'},
```

Skip to main content

```
  '2': 'Douglas Extension'},
 'address.suite': {'0': 'Apt. 556', '1': 'Suite 879', '2': 'Suite 847'},
 'address.city': {'0': 'Gwenborough',
  '1': 'Wisokyburgh',
  '2': 'McKenziehaven'},
 'address.zipcode': {'0': '92998-3874', '1': '90566-7771', '2': '59590-4157'},
 'address.geo.lat': {'0': '-37.3159', '1': '-43.9509', '2': '-68.6102'},
 'address.geo.lng': {'0': '81.1496', '1': '-34.4618', '2': '-47.0653'},
 'company.name': {'0': 'Romaguera-Crona',
  '1': 'Deckow-Crist',
  '2': 'Romaguera-Jacobson'},
 'company.catchPhrase': {'0': 'Multi-layered client-server neural-net',
  '1': 'Proactive didactic contingency',
  '2': 'Face to face bifurcated interface'},
 'company.bs': {'0': 'harness real-time e-markets',
  '1': 'synergize scalable supply-chains',
  '2': 'e-enable strategic applications'}}
```

- `orient="split"` works with JSON files organized as dictionary with three lists: `columns` lists the column names, `index` lists the row names, and `data` is a list-of-lists of data points, one list for each row.

```
new_json = users_df.to_json(orient="split")
json.loads(new_json)
```

```
{'columns': ['id',
  'name',
  'username',
  'email',
  'phone',
  'website',
  'address.street',
  'address.suite',
  'address.city',
  'address.zipcode',
  'address.geo.lat',
  'address.geo.lng',
  'company.name',
  'company.catchPhrase',
  'company.bs'],
 'index': [0, 1, 2],
 'data': [[1,
   'Leanne Graham',
   'Bret',
   'Sincere@april.biz',
   '1-770-736-8031 x56442',
   'hildegard.org',
   'Kulas Light',
   'Apt. 556',
   'Gwenborough',
   '92998-3874',
   '-37.3159',
   '81.1496',
```

Skip to main content

```
  'harness real-time e-markets'],
 [2,
  'Ervin Howell',
  'Antonette',
  'Shanna@melissa.tv',
  '010-692-6593 x09125',
  'anastasia.net',
  'Victor Plains',
  'Suite 879',
  'Wisokyburgh',
  '90566-7771',
  '-43.9509',
  '-34.4618',
  'Deckow-Crist',
  'Proactive didactic contingency',
  'synergize scalable supply-chains'],
 [3,
  'Clementine Bauch',
  'Samantha',
  'Nathan@yesenia.net',
  '1-463-123-4447',
  'ramiro.info',
  'Douglas Extension',
  'Suite 847',
  'McKenziehaven',
  '59590-4157',
  '-68.6102',
  '-47.0653',
  'Romaguera-Jacobson',
  'Face to face bifurcated interface',
  'e-enable strategic applications']]}
```

- `orient="index"` is like `orient="records"` but includes the name of each row in the data:

```
new_json = users_df.to_json(orient="index")
json.loads(new_json)
```

```
{'0': {'id': 1,
  'name': 'Leanne Graham',
  'username': 'Bret',
  'email': 'Sincere@april.biz',
  'phone': '1-770-736-8031 x56442',
  'website': 'hildegard.org',
  'address.street': 'Kulas Light',
  'address.suite': 'Apt. 556',
  'address.city': 'Gwenborough',
  'address.zipcode': '92998-3874',
  'address.geo.lat': '-37.3159',
  'address.geo.lng': '81.1496',
  'company.name': 'Romaguera-Crona',
  'company.catchPhrase': 'Multi-layered client-server neural-net',
  'company.bs': 'harness real-time e-markets'},
 '1': {'id': 2,
  'name': 'Ervin Howell',
  'phone': '010-692-6593 x09125',
  'website': 'anastasia.net',
  'address.street': 'Victor Plains',
  'address.suite': 'Suite 879',
  'address.city': 'Wisokyburgh',
  'address.zipcode': '90566-7771',
  'address.geo.lat': '-43.9509',
  'address.geo.lng': '-34.4618',
  'company.name': 'Deckow-Crist',
  'company.catchPhrase': 'Proactive didactic contingency',
  'company.bs': 'synergize scalable supply-chains'},
 '2': {'id': 3,
  'name': 'Clementine Bauch',
  'username': 'Samantha',
  'email': 'Nathan@yesenia.net',
  'phone': '1-463-123-4447',
  'website': 'ramiro.info',
  'address.street': 'Douglas Extension',
  'address.suite': 'Suite 847',
  'address.city': 'McKenziehaven',
  'address.zipcode': '59590-4157',
  'address.geo.lat': '-68.6102',
  'address.geo.lng': '-47.0653',
  'company.name': 'Romaguera-Jacobson',
  'company.catchPhrase': 'Face to face bifurcated interface',
  'company.bs': 'e-enable strategic applications'}}
```

- `orient="values"` only contains the datapoints:

```
new_json = users_df.to_json(orient="values")
json.loads(new_json)
```

```
[[1,
  'Leanne Graham',
  'Bret',
  'Sincere@april.biz',
  '1-770-736-8031 x56442',
  'hildegard.org',
  'Kulas Light',
  'Apt. 556',
  'Gwenborough',
  '92998-3874',
  '-37.3159',
  '81.1496',
  'Romaguera-Crona',
  'Multi-layered client-server neural-net',
  'harness real-time e-markets'],
 [2,
  'Ervin Howell',
  'Antonette',
  'Shanna@melissa.tv',
  '010-692-6593 x09125',
  'anastasia.net',
  'Victor Plains',
  '90566-7771',
  '-43.9509',
  '-34.4618',
  'Deckow-Crist',
  'Proactive didactic contingency',
  'synergize scalable supply-chains'],
 [3,
  'Clementine Bauch',
  'Samantha',
  'Nathan@yesenia.net',
  '1-463-123-4447',
  'ramiro.info',
  'Douglas Extension',
  'Suite 847',
  'McKenziehaven',
  '59590-4157',
  '-68.6102',
  '-47.0653',
  'Romaguera-Jacobson',
  'Face to face bifurcated interface',
  'e-enable strategic applications']]
```

To save these files to disk, specify a filename (using the `os.chdir()` function to change the working directory to the folder in which you save to save this file), or a filename and path, for the first parameter:

```
users_df.to_json("myjson.json", orient="values")
```