Note NaN's, NaT's and None will be converted to null and datetime objects will be converted based on the date_format and date_unit parameters.

```
In [195]: dfj = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [196]: json = dfj.to_json()

In [197]: json
Out[197]: '{"A":{"0":-1.2945235903,"1":0.2766617129,"2":-0.0139597524,"3":-0.
↪0061535699,"4":0.8957173022},"B":{"0":0.4137381054,"1":-0.472034511,"2":-0.
↪3625429925,"3":-0.923060654,"4":0.8052440254}}'
```

### Orient options

There are a number of different options for the format of the resulting JSON file / string. Consider the following DataFrame and Series:

```
In [198]: dfjo = pd.DataFrame(dict(A=range(1, 4), B=range(4, 7), C=range(7, 10)),
   .....:                      columns=list('ABC'), index=list('xyz'))
   .....:

In [199]: dfjo
Out[199]:
   A  B  C
x  1  4  7
y  2  5  8
z  3  6  9

In [200]: sjo = pd.Series(dict(x=15, y=16, z=17), name='D')

In [201]: sjo
Out[201]:
x    15
y    16
z    17
Name: D, dtype: int64
```

**Column oriented** (the default for DataFrame) serializes the data as nested JSON objects with column labels acting as the primary index:

```
In [202]: dfjo.to_json(orient="columns")
Out[202]: '{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":{"x":7,"y":8,"z":9}}'

# Not available for Series
```

**Index oriented** (the default for Series) similar to column oriented but the index labels are now primary:

```
In [203]: dfjo.to_json(orient="index")
Out[203]: '{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":{"A":3,"B":6,"C":9}}'

In [204]: sjo.to_json(orient="index")
Out[204]: '{"x":15,"y":16,"z":17}'
```

**Record oriented** serializes the data to a JSON array of column -> value records, index labels are not included. This is useful for passing DataFrame data to plotting libraries, for example the JavaScript library d3.js:

```
In [205]: dfjo.to_json(orient="records")
Out[205]: '[{"A":1,"B":4,"C":7},{"A":2,"B":5,"C":8},{"A":3,"B":6,"C":9}]'

In [206]: sjo.to_json(orient="records")
Out[206]: '[15,16,17]'
```

**Value oriented** is a bare-bones option which serializes to nested JSON arrays of values only, column and index labels are not included:

```
In [207]: dfjo.to_json(orient="values")
Out[207]: '[[1,4,7],[2,5,8],[3,6,9]]'

# Not available for Series
```

**Split oriented** serializes to a JSON object containing separate entries for values, index and columns. Name is also included for `Series`:

```
In [208]: dfjo.to_json(orient="split")
Out[208]: '{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,5,8],[3,
↪6,9]]}'

In [209]: sjo.to_json(orient="split")
Out[209]: '{"name":"D","index":["x","y","z"],"data":[15,16,17]}'
```

**Table oriented** serializes to the JSON Table Schema, allowing for the preservation of metadata including but not limited to dtypes and index names.

**Note:** Any orient option that encodes to a JSON object will not preserve the ordering of index and column labels during round-trip serialization. If you wish to preserve label ordering use the *split* option as it uses ordered containers.

### Date handling

Writing in ISO date format:

```
In [210]: dfd = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [211]: dfd['date'] = pd.Timestamp('20130101')

In [212]: dfd = dfd.sort_index(1, ascending=False)

In [213]: json = dfd.to_json(date_format='iso')

In [214]: json
Out[214]: '{"date":{"0":"2013-01-01T00:00:00.000Z","1":"2013-01-01T00:00:00.000Z","2":
↪"2013-01-01T00:00:00.000Z","3":"2013-01-01T00:00:00.000Z","4":"2013-01-01T00:00:00.
↪000Z"},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857,"4
↪":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.
↪4108345112,"4":0.1320031703}}'
```

Writing in ISO date format, with microseconds:

```
In [215]: json = dfd.to_json(date_format='iso', date_unit='us')
```

(continues on next page)

(continued from previous page)

```
In [216]: json
Out[216]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00:00.000000Z
↪","2":"2013-01-01T00:00:00.000000Z","3":"2013-01-01T00:00:00.000000Z","4":"2013-01-
↪01T00:00:00.000000Z"},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":
↪0.8138502857,"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.
↪1702987971,"3":0.4108345112,"4":0.1320031703}}'
```

Epoch timestamps, in seconds:

```
In [217]: json = dfd.to_json(date_format='epoch', date_unit='s')

In [218]: json
Out[218]: '{"date":{"0":1356998400,"1":1356998400,"2":1356998400,"3":1356998400,"4":
↪1356998400},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.
↪8138502857,"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.
↪1702987971,"3":0.4108345112,"4":0.1320031703}}'
```

Writing to a file, with a date index and a date column:

```
In [219]: dfj2 = dfj.copy()

In [220]: dfj2['date'] = pd.Timestamp('20130101')

In [221]: dfj2['ints'] = list(range(5))

In [222]: dfj2['bools'] = True

In [223]: dfj2.index = pd.date_range('20130101', periods=5)

In [224]: dfj2.to_json('test.json')

In [225]: with open('test.json') as fh:
   .....:     print(fh.read())
   .....:
{"A":{"1356998400000":-1.2945235903,"1357084800000":0.2766617129,"1357171200000":-0.
↪0139597524,"1357257600000":-0.0061535699,"1357344000000":0.8957173022},"B":{
↪"1356998400000":0.4137381054,"1357084800000":-0.472034511,"1357171200000":-0.
↪3625429925,"1357257600000":-0.923060654,"1357344000000":0.8052440254},"date":{
↪"1356998400000":1356998400000,"1357084800000":1356998400000,"1357171200000":
↪1356998400000,"1357257600000":1356998400000,"1357344000000":1356998400000},"ints":{
↪"1356998400000":0,"1357084800000":1,"1357171200000":2,"1357257600000":3,
↪"1357344000000":4},"bools":{"1356998400000":true,"1357084800000":true,"1357171200000
↪":true,"1357257600000":true,"1357344000000":true}}
```

### Fallback behavior

If the JSON serializer cannot handle the container contents directly it will fall back in the following manner:

- if the dtype is unsupported (e.g. `np.complex`) then the `default_handler`, if provided, will be called for each value, otherwise an exception is raised.
- if an object is unsupported it will attempt the following:
  - check if the object has defined a `toDict` method and call it. A `toDict` method should return a `dict` which will then be JSON serialized.
  - invoke the `default_handler` if one was provided.

  - convert the object to a `dict` by traversing its contents. However this will often fail with an `OverflowError` or give unexpected results.

In general the best approach for unsupported objects or dtypes is to provide a `default_handler`. For example:

```
>>> DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json()  # raises
RuntimeError: Unhandled numpy dtype 15
```

can be dealt with by specifying a simple `default_handler`:

```
In [226]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
Out[226]: '{"0":{"0":"(1+0j)","1":"(2+0j)","2":"(1+2j)"}}'
```

### Reading JSON

Reading a JSON string to pandas object can take a number of parameters. The parser will try to parse a `DataFrame` if `typ` is not supplied or is `None`. To explicitly force `Series` parsing, pass `typ=series`

- `filepath_or_buffer` : a **VALID** JSON string or file handle / StringIO. The string could be a URL. Valid URL schemes include http, ftp, S3, and file. For file URLs, a host is expected. For instance, a local file could be file ://localhost/path/to/table.json
- `typ` : type of object to recover (series or frame), default 'frame'
- `orient` :

  **Series :**

    - default is `index`
    - allowed values are {`split`, `records`, `index`}

  **DataFrame**

    - default is `columns`
    - allowed values are {`split`, `records`, `index`, `columns`, `values`, `table`}

  The format of the JSON string

| | |
|---|---|
| `split` | dict like {index -> [index], columns -> [columns], data -> [values]} |
| `records` | list like [{column -> value}, ... , {column -> value}] |
| `index` | dict like {index -> {column -> value}} |
| `columns` | dict like {column -> {index -> value}} |
| `values` | just the values array |
| `table` | adhering to the JSON Table Schema |

- `dtype` : if True, infer dtypes, if a dict of column to dtype, then use those, if `False`, then don't infer dtypes at all, default is True, apply only to the data.
- `convert_axes` : boolean, try to convert the axes to the proper dtypes, default is `True`
- `convert_dates` : a list of columns to parse for dates; If `True`, then try to parse date-like columns, default is `True`.
- `keep_default_dates` : boolean, default `True`. If parsing dates, then parse the default date-like columns.
- `numpy` : direct decoding to NumPy arrays. default is `False`; Supports numeric data only, although labels may be non-numeric. Also note that the JSON ordering **MUST** be the same for each term if `numpy=True`.

- `precise_float` : boolean, default `False`. Set to enable usage of higher precision (strtod) function when decoding string to double values. Default (`False`) is to use fast but less precise builtin functionality.
- `date_unit` : string, the timestamp unit to detect if converting dates. Default None. By default the timestamp precision will be detected, if this is not desired then pass one of 's', 'ms', 'us' or 'ns' to force timestamp precision to seconds, milliseconds, microseconds or nanoseconds respectively.
- `lines` : reads file as one json object per line.
- `encoding` : The encoding to use to decode py3 bytes.
- `chunksize` : when used in combination with `lines=True`, return a JsonReader which reads in `chunksize` lines per iteration.

The parser will raise one of `ValueError/TypeError/AssertionError` if the JSON is not parseable.

If a non-default `orient` was used when encoding to JSON be sure to pass the same option here so that decoding produces sensible results, see *Orient Options* for an overview.

### Data conversion

The default of `convert_axes=True`, `dtype=True`, and `convert_dates=True` will try to parse the axes, and all of the data into appropriate types, including dates. If you need to override specific dtypes, pass a dict to `dtype`. `convert_axes` should only be set to `False` if you need to preserve string-like numbers (e.g. '1', '2') in an axes.

---

**Note:** Large integer values may be converted to dates if `convert_dates=True` and the data and / or column labels appear 'date-like'. The exact threshold depends on the `date_unit` specified. 'date-like' means that the column label meets one of the following criteria:

- it ends with `'_at'`
- it ends with `'_time'`
- it begins with `'timestamp'`
- it is `'modified'`
- it is `'date'`

---

**Warning:** When reading JSON data, automatic coercing into dtypes has some quirks:

- an index can be reconstructed in a different order from serialization, that is, the returned order is not guaranteed to be the same as before serialization
- a column that was `float` data will be converted to `integer` if it can be done safely, e.g. a column of `1.`
- bool columns will be converted to `integer` on reconstruction

Thus there are times where you may want to specify specific dtypes via the `dtype` keyword argument.

Reading from a JSON string:

```
In [227]: pd.read_json(json)
Out[227]:
         date         B         A
0 2013-01-01  2.565646 -1.206412
1 2013-01-01  1.340309  1.431256
```

(continues on next page)

(continued from previous page)

```
2 2013-01-01 -0.226169 -1.170299
3 2013-01-01  0.813850  0.410835
4 2013-01-01 -0.827317  0.132003
```

Reading from a file:

```
In [228]: pd.read_json('test.json')
Out[228]:
                   A         B       date  ints  bools
2013-01-01 -1.294524  0.413738 2013-01-01     0   True
2013-01-02  0.276662 -0.472035 2013-01-01     1   True
2013-01-03 -0.013960 -0.362543 2013-01-01     2   True
2013-01-04 -0.006154 -0.923061 2013-01-01     3   True
2013-01-05  0.895717  0.805244 2013-01-01     4   True
```

Don't convert any data (but still convert axes and dates):

```
In [229]: pd.read_json('test.json', dtype=object).dtypes
Out[229]:
A        object
B        object
date     object
ints     object
bools    object
dtype: object
```

Specify dtypes for conversion:

```
In [230]: pd.read_json('test.json', dtype={'A': 'float32', 'bools': 'int8'}).dtypes
Out[230]:
A               float32
B               float64
date     datetime64[ns]
ints              int64
bools              int8
dtype: object
```

Preserve string indices:

```
In [231]: si = pd.DataFrame(np.zeros((4, 4)), columns=list(range(4)),
   .....:                   index=[str(i) for i in range(4)])
   .....:

In [232]: si
Out[232]:
     0    1    2    3
0  0.0  0.0  0.0  0.0
1  0.0  0.0  0.0  0.0
2  0.0  0.0  0.0  0.0
3  0.0  0.0  0.0  0.0

In [233]: si.index
Out[233]: Index(['0', '1', '2', '3'], dtype='object')

In [234]: si.columns
Out[234]: Int64Index([0, 1, 2, 3], dtype='int64')
```

(continues on next page)

(continued from previous page)

```
In [235]: json = si.to_json()

In [236]: sij = pd.read_json(json, convert_axes=False)

In [237]: sij
Out[237]:
   0  1  2  3
0  0  0  0  0
1  0  0  0  0
2  0  0  0  0
3  0  0  0  0

In [238]: sij.index
Out[238]: Index(['0', '1', '2', '3'], dtype='object')

In [239]: sij.columns
Out[239]: Index(['0', '1', '2', '3'], dtype='object')
```

Dates written in nanoseconds need to be read back in nanoseconds:

```
In [240]: json = dfj2.to_json(date_unit='ns')

# Try to parse timestamps as milliseconds -> Won't Work
In [241]: dfju = pd.read_json(json, date_unit='ms')

In [242]: dfju
Out[242]:
                            A         B                   date  ints  bools
1356998400000000000 -1.294524  0.413738  1356998400000000000     0   True
1357084800000000000  0.276662 -0.472035  1356998400000000000     1   True
1357171200000000000 -0.013960 -0.362543  1356998400000000000     2   True
1357257600000000000 -0.006154 -0.923061  1356998400000000000     3   True
1357344000000000000  0.895717  0.805244  1356998400000000000     4   True

# Let pandas detect the correct precision
In [243]: dfju = pd.read_json(json)

In [244]: dfju
Out[244]:
                   A         B       date  ints  bools
2013-01-01 -1.294524  0.413738 2013-01-01     0   True
2013-01-02  0.276662 -0.472035 2013-01-01     1   True
2013-01-03 -0.013960 -0.362543 2013-01-01     2   True
2013-01-04 -0.006154 -0.923061 2013-01-01     3   True
2013-01-05  0.895717  0.805244 2013-01-01     4   True

# Or specify that all timestamps are in nanoseconds
In [245]: dfju = pd.read_json(json, date_unit='ns')

In [246]: dfju
Out[246]:
                   A         B       date  ints  bools
2013-01-01 -1.294524  0.413738 2013-01-01     0   True
2013-01-02  0.276662 -0.472035 2013-01-01     1   True
2013-01-03 -0.013960 -0.362543 2013-01-01     2   True
2013-01-04 -0.006154 -0.923061 2013-01-01     3   True
```

(continues on next page)

(continued from previous page)

```
2013-01-05  0.895717  0.805244 2013-01-01     4   True
```

### The Numpy parameter

**Note:** This param has been deprecated as of version 1.0.0 and will raise a `FutureWarning`.

This supports numeric data only. Index and columns labels may be non-numeric, e.g. strings, dates etc.

If `numpy=True` is passed to `read_json` an attempt will be made to sniff an appropriate dtype during deserialization and to subsequently decode directly to NumPy arrays, bypassing the need for intermediate Python objects.

This can provide speedups if you are deserialising a large amount of numeric data:

```
In [247]: randfloats = np.random.uniform(-100, 1000, 10000)

In [248]: randfloats.shape = (1000, 10)

In [249]: dffloats = pd.DataFrame(randfloats, columns=list('ABCDEFGHIJ'))

In [250]: jsonfloats = dffloats.to_json()
```

```
In [251]: %timeit pd.read_json(jsonfloats)
20.4 ms +- 2.45 ms per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [252]: %timeit pd.read_json(jsonfloats, numpy=True)
15.8 ms +- 1.18 ms per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

The speedup is less noticeable for smaller datasets:

```
In [253]: jsonfloats = dffloats.head(100).to_json()
```

```
In [254]: %timeit pd.read_json(jsonfloats)
11.3 ms +- 1.04 ms per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [255]: %timeit pd.read_json(jsonfloats, numpy=True)
11.2 ms +- 1.85 ms per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

**Warning:** Direct NumPy decoding makes a number of assumptions and may fail or produce unexpected output if these assumptions are not satisfied:

- data is numeric.
- data is uniform. The dtype is sniffed from the first value decoded. A `ValueError` may be raised, or incorrect output may be produced if this condition is not satisfied.
- labels are ordered. Labels are only read from the first container, it is assumed that each subsequent row / column has been encoded in the same order. This should be satisfied if the data was encoded using `to_json` but may not be the case if the JSON is from another source.

### Normalization

pandas provides a utility function to take a dict or list of dicts and *normalize* this semi-structured data into a flat table.

```
In [256]: data = [{'id': 1, 'name': {'first': 'Coleen', 'last': 'Volk'}},
   .....:         {'name': {'given': 'Mose', 'family': 'Regner'}},
   .....:         {'id': 2, 'name': 'Faye Raker'}]
   .....:

In [257]: pd.json_normalize(data)
Out[257]:
    id name.first name.last name.given name.family        name
0  1.0     Coleen      Volk        NaN         NaN         NaN
1  NaN        NaN       NaN       Mose      Regner         NaN
2  2.0        NaN       NaN        NaN         NaN  Faye Raker
```

```
In [258]: data = [{'state': 'Florida',
   .....:          'shortname': 'FL',
   .....:          'info': {'governor': 'Rick Scott'},
   .....:          'county': [{'name': 'Dade', 'population': 12345},
   .....:                     {'name': 'Broward', 'population': 40000},
   .....:                     {'name': 'Palm Beach', 'population': 60000}]},
   .....:         {'state': 'Ohio',
   .....:          'shortname': 'OH',
   .....:          'info': {'governor': 'John Kasich'},
   .....:          'county': [{'name': 'Summit', 'population': 1234},
   .....:                     {'name': 'Cuyahoga', 'population': 1337}]}]
   .....:

In [259]: pd.json_normalize(data, 'county', ['state', 'shortname', ['info', 'governor
↪']])
Out[259]:
         name  population    state shortname info.governor
0        Dade       12345  Florida        FL    Rick Scott
1     Broward       40000  Florida        FL    Rick Scott
2  Palm Beach       60000  Florida        FL    Rick Scott
3      Summit        1234     Ohio        OH   John Kasich
4    Cuyahoga        1337     Ohio        OH   John Kasich
```

The max_level parameter provides more control over which level to end normalization. With max_level=1 the following snippet normalizes until 1st nesting level of the provided dict.

```
In [260]: data = [{'CreatedBy': {'Name': 'User001'},
   .....:          'Lookup': {'TextField': 'Some text',
   .....:                     'UserField': {'Id': 'ID001',
   .....:                                   'Name': 'Name001'}},
   .....:          'Image': {'a': 'b'}
   .....:          }]
   .....:

In [261]: pd.json_normalize(data, max_level=1)
Out[261]:
  CreatedBy.Name Lookup.TextField                    Lookup.UserField Image.a
0        User001        Some text  {'Id': 'ID001', 'Name': 'Name001'}       b
```

### Line delimited json

pandas is able to read and write line-delimited json files that are common in data processing pipelines using Hadoop or Spark.

New in version 0.21.0.

For line-delimited json files, pandas can also return an iterator which reads in `chunksize` lines at a time. This can be useful for large files or to read from a stream.

```
In [262]: jsonl = '''
   .....:     {"a": 1, "b": 2}
   .....:     {"a": 3, "b": 4}
   .....: '''
   .....:

In [263]: df = pd.read_json(jsonl, lines=True)

In [264]: df
Out[264]:
   a  b
0  1  2
1  3  4

In [265]: df.to_json(orient='records', lines=True)
Out[265]: '{"a":1,"b":2}\n{"a":3,"b":4}'

# reader is an iterator that returns `chunksize` lines each iteration
In [266]: reader = pd.read_json(StringIO(jsonl), lines=True, chunksize=1)

In [267]: reader
Out[267]: <pandas.io.json._json.JsonReader at 0x7f5336d04d10>

In [268]: for chunk in reader:
   .....:     print(chunk)
   .....:
Empty DataFrame
Columns: []
Index: []
   a  b
0  1  2
   a  b
1  3  4
```

### Table schema

Table Schema is a spec for describing tabular datasets as a JSON object. The JSON includes information on the field names, types, and other attributes. You can use the orient `table` to build a JSON string with two fields, `schema` and `data`.

```
In [269]: df = pd.DataFrame({'A': [1, 2, 3],
   .....:                    'B': ['a', 'b', 'c'],
   .....:                    'C': pd.date_range('2016-01-01', freq='d', periods=3)},
   .....:                   index=pd.Index(range(3), name='idx'))
   .....:

In [270]: df
```

(continues on next page)

(continued from previous page)

```
Out[270]:
     A  B          C
idx
0    1  a 2016-01-01
1    2  b 2016-01-02
2    3  c 2016-01-03

In [271]: df.to_json(orient='table', date_format="iso")
Out[271]: '{"schema":{"fields":[{"name":"idx","type":"integer"},{"name":"A","type":
↪"integer"},{"name":"B","type":"string"},{"name":"C","type":"datetime"}],"primaryKey
↪":["idx"],"pandas_version":"0.20.0"},"data":[{"idx":0,"A":1,"B":"a","C":"2016-01-
↪01T00:00:00.000Z"},{"idx":1,"A":2,"B":"b","C":"2016-01-02T00:00:00.000Z"},{"idx":2,
↪"A":3,"B":"c","C":"2016-01-03T00:00:00.000Z"}]}'
```

The `schema` field contains the `fields` key, which itself contains a list of column name to type pairs, including the `Index` or `MultiIndex` (see below for a list of types). The `schema` field also contains a `primaryKey` field if the (Multi)index is unique.

The second field, `data`, contains the serialized data with the `records` orient. The index is included, and any datetimes are ISO 8601 formatted, as required by the Table Schema spec.

The full list of types supported are described in the Table Schema spec. This table shows the mapping from pandas types:

| Pandas type | Table Schema type |
|---|---|
| int64 | integer |
| float64 | number |
| bool | boolean |
| datetime64[ns] | datetime |
| timedelta64[ns] | duration |
| categorical | any |
| object | str |

A few notes on the generated table schema:

- The `schema` object contains a `pandas_version` field. This contains the version of pandas' dialect of the schema, and will be incremented with each revision.
- All dates are converted to UTC when serializing. Even timezone naive values, which are treated as UTC with an offset of 0.

```
In [272]: from pandas.io.json import build_table_schema

In [273]: s = pd.Series(pd.date_range('2016', periods=4))

In [274]: build_table_schema(s)
Out[274]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'datetime'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- datetimes with a timezone (before serializing), include an additional field `tz` with the time zone name (e.g. `'US/Central'`).

```
In [275]: s_tz = pd.Series(pd.date_range('2016', periods=12,
   .....:                                tz='US/Central'))
   .....:

In [276]: build_table_schema(s_tz)
Out[276]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'datetime', 'tz': 'US/Central'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- Periods are converted to timestamps before serialization, and so have the same behavior of being converted to UTC. In addition, periods will contain and additional field `freq` with the period's frequency, e.g. `'A-DEC'`.

```
In [277]: s_per = pd.Series(1, index=pd.period_range('2016', freq='A-DEC',
   .....:                                            periods=4))
   .....:

In [278]: build_table_schema(s_per)
Out[278]:
{'fields': [{'name': 'index', 'type': 'datetime', 'freq': 'A-DEC'},
  {'name': 'values', 'type': 'integer'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- Categoricals use the `any` type and an `enum` constraint listing the set of possible values. Additionally, an `ordered` field is included:

```
In [279]: s_cat = pd.Series(pd.Categorical(['a', 'b', 'a']))

In [280]: build_table_schema(s_cat)
Out[280]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values',
   'type': 'any',
   'constraints': {'enum': ['a', 'b']},
   'ordered': False}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- A `primaryKey` field, containing an array of labels, is included *if the index is unique*:

```
In [281]: s_dupe = pd.Series([1, 2], index=[1, 1])

In [282]: build_table_schema(s_dupe)
Out[282]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
 'pandas_version': '0.20.0'}
```

- The `primaryKey` behavior is the same with MultiIndexes, but in this case the `primaryKey` is an array:

```
In [283]: s_multi = pd.Series(1, index=pd.MultiIndex.from_product([('a', 'b'),
   .....:                                                          (0, 1)]))
   .....:

In [284]: build_table_schema(s_multi)
```

(continues on next page)

(continued from previous page)

```
Out[284]:
{'fields': [{'name': 'level_0', 'type': 'string'},
  {'name': 'level_1', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
 'primaryKey': FrozenList(['level_0', 'level_1']),
 'pandas_version': '0.20.0'}
```

- The default naming roughly follows these rules:
  - For series, the `object.name` is used. If that's none, then the name is `values`
  - For `DataFrames`, the stringified version of the column name is used
  - For `Index` (not `MultiIndex`), `index.name` is used, with a fallback to `index` if that is None.
  - For `MultiIndex`, `mi.names` is used. If any level has no name, then `level_<i>` is used.

New in version 0.23.0.

`read_json` also accepts `orient='table'` as an argument. This allows for the preservation of metadata such as dtypes and index names in a round-trippable manner.

```
In [285]: df = pd.DataFrame({'foo': [1, 2, 3, 4],
   .....:                    'bar': ['a', 'b', 'c', 'd'],
   .....:                    'baz': pd.date_range('2018-01-01', freq='d', periods=4),
   .....:                    'qux': pd.Categorical(['a', 'b', 'c', 'c'])
   .....:                    }, index=pd.Index(range(4), name='idx'))
   .....:

In [286]: df
Out[286]:
     foo bar        baz qux
idx
0      1   a 2018-01-01   a
1      2   b 2018-01-02   b
2      3   c 2018-01-03   c
3      4   d 2018-01-04   c

In [287]: df.dtypes
Out[287]:
foo              int64
bar             object
baz    datetime64[ns]
qux          category
dtype: object

In [288]: df.to_json('test.json', orient='table')

In [289]: new_df = pd.read_json('test.json', orient='table')

In [290]: new_df
Out[290]:
     foo bar        baz qux
idx
0      1   a 2018-01-01   a
1      2   b 2018-01-02   b
2      3   c 2018-01-03   c
3      4   d 2018-01-04   c
```

(continues on next page)

(continued from previous page)

```
In [291]: new_df.dtypes
Out[291]:
foo              int64
bar             object
baz     datetime64[ns]
qux           category
dtype: object
```

Please note that the literal string 'index' as the name of an *Index* is not round-trippable, nor are any names beginning with `'level_'` within a *MultiIndex*. These are used by default in *DataFrame.to_json()* to indicate missing values and the subsequent read cannot distinguish the intent.

```
In [292]: df.index.name = 'index'

In [293]: df.to_json('test.json', orient='table')

In [294]: new_df = pd.read_json('test.json', orient='table')

In [295]: print(new_df.index.name)
None
```

### 2.1.3 HTML

#### Reading HTML content

**Warning:** We **highly encourage** you to read the *HTML Table Parsing gotchas* below regarding the issues surrounding the BeautifulSoup4/html5lib/lxml parsers.

The top-level `read_html()` function can accept an HTML string/file/URL and will parse HTML tables into list of pandas `DataFrames`. Let's look at a few examples.

**Note:** `read_html` returns a `list` of `DataFrame` objects, even if there is only a single table contained in the HTML content.

Read a URL with no options:

```
In [296]: url = 'https://www.fdic.gov/bank/individual/failed/banklist.html'

In [297]: dfs = pd.read_html(url)

In [298]: dfs
Out[298]:
[                            Bank Name           City  ST   CERT
↪Acquiring Institution       Closing Date
 0                The First State Bank  Barboursville  WV  14361
↪  MVB Bank, Inc.      April 3, 2020
 1                  Ericson State Bank        Ericson  NE  18265           Farmers
↪and Merchants Bank  February 14, 2020
 2     City National Bank of New Jersey         Newark  NJ  21111
↪ Industrial Bank   November 1, 2019
```

(continues on next page)

(continued from previous page)

```
 3                      Resolute Bank         Maumee  OH  58317                    ␣
↪Buckeye State Bank   October 25, 2019
 4                Louisa Community Bank         Louisa  KY  58112    Kentucky Farmers␣
↪Bank Corporation   October 25, 2019
 ..                                ...            ...  ..    ...                    ␣
↪             ...                  ...
 556                 Superior Bank, FSB       Hinsdale  IL  32646                 ␣
↪Superior Federal, FSB      July 27, 2001
 557               Malta National Bank          Malta  OH   6629                    ␣
↪North Valley Bank        May 3, 2001
 558    First Alliance Bank & Trust Co.     Manchester  NH  34264  Southern New␣
↪Hampshire Bank & Trust   February 2, 2001
 559  National State Bank of Metropolis     Metropolis  IL   3815                ␣
↪Banterra Bank of Marion  December 14, 2000
 560                   Bank of Honolulu       Honolulu  HI  21029                    ␣
↪Bank of the Orient   October 13, 2000

 [561 rows x 6 columns]]
```

**Note:** The data from the above URL changes every Monday so the resulting data above and the data below may be slightly different.

Read in the content of the file from the above URL and pass it to `read_html` as a string:

```
In [299]: with open(file_path, 'r') as f:
   .....:     dfs = pd.read_html(f.read())
   .....:

In [300]: dfs
Out[300]:
[                                 Bank Name           City  ST  ...                ␣
↪Acquiring Institution       Closing Date       Updated Date
 0    Banks of Wisconsin d/b/a Bank of Kenosha       Kenosha  WI  ...                ␣
↪North Shore Bank, FSB       May 31, 2013       May 31, 2013
 1                      Central Arizona Bank    Scottsdale  AZ  ...                ␣
↪  Western State Bank       May 14, 2013       May 20, 2013
 2                           Sunrise Bank      Valdosta  GA  ...                ␣
↪        Synovus Bank       May 10, 2013       May 21, 2013
 3                     Pisgah Community Bank     Asheville  NC  ...                ␣
↪  Capital Bank, N.A.       May 10, 2013       May 14, 2013
 4                       Douglas County Bank  Douglasville  GA  ...                ␣
↪ Hamilton State Bank     April 26, 2013       May 16, 2013
 ..                                    ...           ...  ..  ...                ␣
↪                ...                  ...           ...
 500                     Superior Bank, FSB      Hinsdale  IL  ...                ␣
↪Superior Federal, FSB      July 27, 2001       June 5, 2012
 501                    Malta National Bank         Malta  OH  ...                ␣
↪   North Valley Bank        May 3, 2001  November 18, 2002
 502          First Alliance Bank & Trust Co.    Manchester  NH  ...  Southern New␣
↪Hampshire Bank & Trust   February 2, 2001  February 18, 2003
 503        National State Bank of Metropolis    Metropolis  IL  ...             ␣
↪Banterra Bank of Marion  December 14, 2000     March 17, 2005
 504                       Bank of Honolulu      Honolulu  HI  ...                ␣
↪  Bank of the Orient   October 13, 2000     March 17, 2005
```

(continues on next page)

(continued from previous page)

```
 [505 rows x 7 columns]]
```

You can even pass in an instance of `StringIO` if you so desire:

```
In [301]: with open(file_path, 'r') as f:
   .....:     sio = StringIO(f.read())
   .....:

In [302]: dfs = pd.read_html(sio)

In [303]: dfs
Out[303]:
[                                 Bank Name          City  ST  ...
↪Acquiring Institution       Closing Date       Updated Date
 0    Banks of Wisconsin d/b/a Bank of Kenosha       Kenosha  WI  ...
↪North Shore Bank, FSB       May 31, 2013       May 31, 2013
 1                         Central Arizona Bank    Scottsdale  AZ  ...
↪  Western State Bank       May 14, 2013       May 20, 2013
 2                              Sunrise Bank      Valdosta  GA  ...
↪        Synovus Bank       May 10, 2013       May 21, 2013
 3                       Pisgah Community Bank     Asheville  NC  ...
↪  Capital Bank, N.A.       May 10, 2013       May 14, 2013
 4                         Douglas County Bank  Douglasville  GA  ...
↪ Hamilton State Bank     April 26, 2013       May 16, 2013
 ..                                     ...           ...  ..  ...
↪                  ...                ...           ...
 500                       Superior Bank, FSB      Hinsdale  IL  ...
↪Superior Federal, FSB      July 27, 2001       June 5, 2012
 501                      Malta National Bank         Malta  OH  ...
↪   North Valley Bank        May 3, 2001  November 18, 2002
 502           First Alliance Bank & Trust Co.    Manchester  NH  ...  Southern New
↪Hampshire Bank & Trust   February 2, 2001  February 18, 2003
 503         National State Bank of Metropolis    Metropolis  IL  ...
↪Banterra Bank of Marion  December 14, 2000     March 17, 2005
 504                           Bank of Honolulu      Honolulu  HI  ...
↪  Bank of the Orient   October 13, 2000     March 17, 2005

 [505 rows x 7 columns]]
```

**Note:** The following examples are not run by the IPython evaluator due to the fact that having so many network-accessing functions slows down the documentation build. If you spot an error or an example that doesn't run, please do not hesitate to report it over on pandas GitHub issues page.

Read a URL and match a table that contains specific text:

```
match = 'Metcalf Bank'
df_list = pd.read_html(url, match=match)
```

Specify a header row (by default <th> or <td> elements located within a <thead> are used to form the column index, if multiple rows are contained within <thead> then a MultiIndex is created); if specified, the header row is taken from the data minus the parsed header elements (<th> elements).

```
dfs = pd.read_html(url, header=0)
```

Specify an index column:

```
dfs = pd.read_html(url, index_col=0)
```

Specify a number of rows to skip:

```
dfs = pd.read_html(url, skiprows=0)
```

Specify a number of rows to skip using a list (`xrange` (Python 2 only) works as well):

```
dfs = pd.read_html(url, skiprows=range(2))
```

Specify an HTML attribute:

```
dfs1 = pd.read_html(url, attrs={'id': 'table'})
dfs2 = pd.read_html(url, attrs={'class': 'sortable'})
print(np.array_equal(dfs1[0], dfs2[0]))  # Should be True
```

Specify values that should be converted to NaN:

```
dfs = pd.read_html(url, na_values=['No Acquirer'])
```

Specify whether to keep the default set of NaN values:

```
dfs = pd.read_html(url, keep_default_na=False)
```

Specify converters for columns. This is useful for numerical text data that has leading zeros. By default columns that are numerical are cast to numeric types and the leading zeros are lost. To avoid this, we can convert these columns to strings.

```
url_mcc = 'https://en.wikipedia.org/wiki/Mobile_country_code'
dfs = pd.read_html(url_mcc, match='Telekom Albania', header=0,
                   converters={'MNC': str})
```

Use some combination of the above:

```
dfs = pd.read_html(url, match='Metcalf Bank', index_col=0)
```

Read in pandas `to_html` output (with some loss of floating point precision):

```
df = pd.DataFrame(np.random.randn(2, 2))
s = df.to_html(float_format='{0:.40g}'.format)
dfin = pd.read_html(s, index_col=0)
```

The `lxml` backend will raise an error on a failed parse if that is the only parser you provide. If you only have a single parser you can provide just a string, but it is considered good practice to pass a list with one string if, for example, the function expects a sequence of strings. You may use:

```
dfs = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml'])
```

Or you could pass `flavor='lxml'` without a list:

```
dfs = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor='lxml')
```

However, if you have bs4 and html5lib installed and pass `None` or `['lxml', 'bs4']` then the parse will most likely succeed. Note that *as soon as a parse succeeds, the function will return*.

```
dfs = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml', 'bs4'])
```

### Writing to HTML files

`DataFrame` objects have an instance method `to_html` which renders the contents of the `DataFrame` as an HTML table. The function arguments are as in the method `to_string` described above.

**Note:** Not all of the possible options for `DataFrame.to_html` are shown here for brevity's sake. See `to_html()` for the full set of options.

```
In [304]: df = pd.DataFrame(np.random.randn(2, 2))

In [305]: df
Out[305]:
          0         1
0 -0.184744  0.496971
1 -0.856240  1.857977

In [306]: print(df.to_html())  # raw html
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

HTML:

The `columns` argument will limit the columns shown:

```
In [307]: print(df.to_html(columns=[0]))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
    </tr>
```

(continues on next page)

(continued from previous page)

```
    <tr>
      <th>1</th>
      <td>-0.856240</td>
    </tr>
  </tbody>
</table>
```

HTML:

`float_format` takes a Python callable to control the precision of floating point values:

```
In [308]: print(df.to_html(float_format='{0:.10f}'.format))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.1847438576</td>
      <td>0.4969711327</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.8562396763</td>
      <td>1.8579766508</td>
    </tr>
  </tbody>
</table>
```

HTML:

`bold_rows` will make the row labels bold by default, but you can turn that off:

```
In [309]: print(df.to_html(bold_rows=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>0</td>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <td>1</td>
      <td>-0.856240</td>
      <td>1.857977</td>
```

(continues on next page)

(continued from previous page)

```
    </tr>
  </tbody>
</table>
```

The `classes` argument provides the ability to give the resulting HTML table CSS classes. Note that these classes are *appended* to the existing `'dataframe'` class.

```
In [310]: print(df.to_html(classes=['awesome_table_class', 'even_more_awesome_class
↪'']))
<table border="1" class="dataframe awesome_table_class even_more_awesome_class">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

The `render_links` argument provides the ability to add hyperlinks to cells that contain URLs.

New in version 0.24.

```
In [311]: url_df = pd.DataFrame({
   .....:     'name': ['Python', 'Pandas'],
   .....:     'url': ['https://www.python.org/', 'https://pandas.pydata.org']})
   .....:

In [312]: print(url_df.to_html(render_links=True))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>name</th>
      <th>url</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>Python</td>
      <td><a href="https://www.python.org/" target="_blank">https://www.python.org/</
↪a></td>
    </tr>
    <tr>
```

(continues on next page)

(continued from previous page)

```
      <th>1</th>
      <td>Pandas</td>
      <td><a href="https://pandas.pydata.org" target="_blank">https://pandas.pydata.
↪org</a></td>
    </tr>
  </tbody>
</table>
```

HTML:

Finally, the `escape` argument allows you to control whether the "<", ">" and "&" characters escaped in the resulting HTML (by default it is `True`). So to get the HTML without escaped characters pass `escape=False`

```
In [313]: df = pd.DataFrame({'a': list('&<>'), 'b': np.random.randn(3)})
```

Escaped:

```
In [314]: print(df.to_html())
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>&amp;</td>
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>
      <td>&lt;</td>
      <td>-0.230305</td>
    </tr>
    <tr>
      <th>2</th>
      <td>&gt;</td>
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>
```

Not escaped:

```
In [315]: print(df.to_html(escape=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
```

(continues on next page)

(continued from previous page)

```
    <tr>
      <th>0</th>
      <td>&</td>
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>
      <td><</td>
      <td>-0.230305</td>
    </tr>
    <tr>
      <th>2</th>
      <td>></td>
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>
```

**Note:** Some browsers may not show a difference in the rendering of the previous two HTML tables.

### HTML Table Parsing Gotchas

There are some versioning issues surrounding the libraries that are used to parse HTML tables in the top-level pandas io function `read_html`.

**Issues with lxml**

- Benefits
  - **lxml** is very fast.
  - **lxml** requires Cython to install correctly.
- Drawbacks
  - **lxml** does *not* make any guarantees about the results of its parse *unless* it is given **strictly valid markup**.
  - In light of the above, we have chosen to allow you, the user, to use the **lxml** backend, but **this backend will use html5lib** if **lxml** fails to parse
  - It is therefore *highly recommended* that you install both **BeautifulSoup4** and **html5lib**, so that you will still get a valid result (provided everything else is valid) even if **lxml** fails.

**Issues with BeautifulSoup4 using lxml as a backend**

- The above issues hold here as well since **BeautifulSoup4** is essentially just a wrapper around a parser backend.

**Issues with BeautifulSoup4 using html5lib as a backend**

- Benefits
  - **html5lib** is far more lenient than **lxml** and consequently deals with *real-life markup* in a much saner way rather than just, e.g., dropping an element without notifying you.
  - **html5lib** *generates valid HTML5 markup from invalid markup automatically*. This is extremely important for parsing HTML tables, since it guarantees a valid document. However, that does NOT mean that it is "correct", since the process of fixing markup does not have a single definition.
  - **html5lib** is pure Python and requires no additional build steps beyond its own installation.

- Drawbacks
  - The biggest drawback to using **html5lib** is that it is slow as molasses. However consider the fact that many tables on the web are not big enough for the parsing algorithm runtime to matter. It is more likely that the bottleneck will be in the process of reading the raw text from the URL over the web, i.e., IO (input-output). For very large tables, this might not be true.

### 2.1.4 Excel files

The *read_excel()* method can read Excel 2003 (`.xls`) files using the `xlrd` Python module. Excel 2007+ (`.xlsx`) files can be read using either `xlrd` or `openpyxl`. Binary Excel (`.xlsb`) files can be read using `pyxlsb`. The *to_excel()* instance method is used for saving a `DataFrame` to Excel. Generally the semantics are similar to working with *csv* data. See the *cookbook* for some advanced strategies.

#### Reading Excel files

In the most basic use-case, `read_excel` takes a path to an Excel file, and the `sheet_name` indicating which sheet to parse.

```
# Returns a DataFrame
pd.read_excel('path_to_file.xls', sheet_name='Sheet1')
```

##### `ExcelFile` class

To facilitate working with multiple sheets from the same file, the `ExcelFile` class can be used to wrap the file and can be passed into `read_excel` There will be a performance benefit for reading multiple sheets as the file is read into memory only once.

```
xlsx = pd.ExcelFile('path_to_file.xls')
df = pd.read_excel(xlsx, 'Sheet1')
```

The `ExcelFile` class can also be used as a context manager.

```
with pd.ExcelFile('path_to_file.xls') as xls:
    df1 = pd.read_excel(xls, 'Sheet1')
    df2 = pd.read_excel(xls, 'Sheet2')
```

The `sheet_names` property will generate a list of the sheet names in the file.

The primary use-case for an `ExcelFile` is parsing multiple sheets with different parameters:

```
data = {}
# For when Sheet1's format differs from Sheet2
with pd.ExcelFile('path_to_file.xls') as xls:
    data['Sheet1'] = pd.read_excel(xls, 'Sheet1', index_col=None,
                                   na_values=['NA'])
    data['Sheet2'] = pd.read_excel(xls, 'Sheet2', index_col=1)
```

Note that if the same parsing parameters are used for all sheets, a list of sheet names can simply be passed to `read_excel` with no loss in performance.

```
# using the ExcelFile class
data = {}
with pd.ExcelFile('path_to_file.xls') as xls:
    data['Sheet1'] = pd.read_excel(xls, 'Sheet1', index_col=None,
                                   na_values=['NA'])
    data['Sheet2'] = pd.read_excel(xls, 'Sheet2', index_col=None,
                                   na_values=['NA'])

# equivalent using the read_excel function
data = pd.read_excel('path_to_file.xls', ['Sheet1', 'Sheet2'],
                     index_col=None, na_values=['NA'])
```

`ExcelFile` can also be called with a `xlrd.book.Book` object as a parameter. This allows the user to control how the excel file is read. For example, sheets can be loaded on demand by calling `xlrd.open_workbook()` with `on_demand=True`.

```
import xlrd
xlrd_book = xlrd.open_workbook('path_to_file.xls', on_demand=True)
with pd.ExcelFile(xlrd_book) as xls:
    df1 = pd.read_excel(xls, 'Sheet1')
    df2 = pd.read_excel(xls, 'Sheet2')
```

### Specifying sheets

**Note:** The second argument is `sheet_name`, not to be confused with `ExcelFile.sheet_names`.

**Note:** An ExcelFile's attribute `sheet_names` provides access to a list of sheets.

- The arguments `sheet_name` allows specifying the sheet or sheets to read.
- The default value for `sheet_name` is 0, indicating to read the first sheet
- Pass a string to refer to the name of a particular sheet in the workbook.
- Pass an integer to refer to the index of a sheet. Indices follow Python convention, beginning at 0.
- Pass a list of either strings or integers, to return a dictionary of specified sheets.
- Pass a `None` to return a dictionary of all available sheets.

```
# Returns a DataFrame
pd.read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

Using the sheet index:

```
# Returns a DataFrame
pd.read_excel('path_to_file.xls', 0, index_col=None, na_values=['NA'])
```

Using all default values:

```
# Returns a DataFrame
pd.read_excel('path_to_file.xls')
```

Using None to get all sheets:

```
# Returns a dictionary of DataFrames
pd.read_excel('path_to_file.xls', sheet_name=None)
```

Using a list to get multiple sheets:

```
# Returns the 1st and 4th sheet, as a dictionary of DataFrames.
pd.read_excel('path_to_file.xls', sheet_name=['Sheet1', 3])
```

`read_excel` can read more than one sheet, by setting `sheet_name` to either a list of sheet names, a list of sheet positions, or `None` to read all sheets. Sheets can be specified by sheet index or sheet name, using an integer or string, respectively.

### Reading a `MultiIndex`

`read_excel` can read a `MultiIndex` index, by passing a list of columns to `index_col` and a `MultiIndex` column by passing a list of rows to `header`. If either the `index` or `columns` have serialized level names those will be read in as well by specifying the rows/columns that make up the levels.

For example, to read in a `MultiIndex` index without names:

```
In [316]: df = pd.DataFrame({'a': [1, 2, 3, 4], 'b': [5, 6, 7, 8]},
   .....:                   index=pd.MultiIndex.from_product([['a', 'b'], ['c', 'd
↪']]))
   .....:

In [317]: df.to_excel('path_to_file.xlsx')

In [318]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1])

In [319]: df
Out[319]:
      a  b
a c   1  5
  d   2  6
b c   3  7
  d   4  8
```

If the index has level names, they will parsed as well, using the same parameters.

```
In [320]: df.index = df.index.set_names(['lvl1', 'lvl2'])

In [321]: df.to_excel('path_to_file.xlsx')

In [322]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1])

In [323]: df
Out[323]:
            a  b
lvl1 lvl2
a    c      1  5
     d      2  6
b    c      3  7
     d      4  8
```

If the source file has both `MultiIndex` index and columns, lists specifying each should be passed to `index_col` and `header`: