sheet_name [str, int, list, or None, default 0] Strings are used for sheet names. Integers are used in zero-indexed sheet positions. Lists of strings/integers are used to request multiple sheets. Specify None to get all sheets.

Available cases:

- Defaults to `0`: 1st sheet as a *DataFrame*
- `1`: 2nd sheet as a *DataFrame*
- `"Sheet1"`: Load sheet with name "Sheet1"
- `[0, 1, "Sheet5"]`: Load first, second and sheet named "Sheet5" as a dict of *DataFrame*
- None: All sheets.

header [int, list of int, default 0] Row (0-indexed) to use for the column labels of the parsed DataFrame. If a list of integers is passed those row positions will be combined into a `MultiIndex`. Use None if there is no header.

names [array-like, default None] List of column names to use. If file contains no header row, then you should explicitly pass header=None.

index_col [int, list of int, default None] Column (0-indexed) to use as the row labels of the DataFrame. Pass None if there is no such column. If a list is passed, those columns will be combined into a `MultiIndex`. If a subset of data is selected with `usecols`, index_col is based on the subset.

usecols [int, str, list-like, or callable default None]

- If None, then parse all columns.
- If str, then indicates comma separated list of Excel column letters and column ranges (e.g. "A:E" or "A,C,E:F"). Ranges are inclusive of both sides.
- If list of int, then indicates list of column numbers to be parsed.
- If list of string, then indicates list of column names to be parsed.

  New in version 0.24.0.
- If callable, then evaluate each column name against it and parse the column if the callable returns `True`.

Returns a subset of the columns according to behavior above.

New in version 0.24.0.

squeeze [bool, default False] If the parsed data only contains one column then return a Series.

dtype [Type name or dict of column -> type, default None] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} Use *object* to preserve data as stored in Excel and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

engine [str, default None] If io is not a buffer or path, this must be set to identify io. Acceptable values are None, "xlrd", "openpyxl" or "odf".

converters [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels, values are functions that take one input argument, the Excel cell content, and return the transformed content.

true_values [list, default None] Values to consider as True.

false_values [list, default None] Values to consider as False.

skiprows [list-like] Rows to skip at the beginning (0-indexed).

nrows [int, default None] Number of rows to parse.

New in version 0.23.0.

na_values [scalar, str, list-like, or dict, default None] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', '<NA>', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.

keep_default_na [bool, default True] Whether or not to include the default NaN values when parsing the data. Depending on whether *na_values* is passed in, the behavior is as follows:

- If *keep_default_na* is True, and *na_values* are specified, *na_values* is appended to the default NaN values used for parsing.
- If *keep_default_na* is True, and *na_values* are not specified, only the default NaN values are used for parsing.
- If *keep_default_na* is False, and *na_values* are specified, only the NaN values specified *na_values* are used for parsing.
- If *keep_default_na* is False, and *na_values* are not specified, no strings will be parsed as NaN.

Note that if *na_filter* is passed in as False, the *keep_default_na* and *na_values* parameters will be ignored.

na_filter [bool, default True] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing na_filter=False can improve the performance of reading a large file.

verbose [bool, default False] Indicate number of NA values placed in non-numeric columns.

parse_dates [bool, list-like, or dict, default False] The behavior is as follows:

- bool. If True -> try parsing the index.
- list of int or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

If a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. If you don`t want to parse some cells as date just change their type in Excel to "Text". For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_excel`.

Note: A fast-path exists for iso8601-formatted dates.

date_parser [function, optional] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call *date_parser* in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by *parse_dates*) as arguments; 2) concatenate (row-wise) the string values from the columns defined by *parse_dates* into a single array and pass that; and 3) call *date_parser* once for each row using one or more strings (corresponding to the columns defined by *parse_dates*) as arguments.

thousands [str, default None] Thousands separator for parsing string columns to numeric. Note that this parameter is only necessary for columns stored as TEXT in Excel, any numeric columns will automatically be parsed, regardless of display format.

comment [str, default None] Comments out remainder of line. Pass a character or characters to this argument to indicate comments in the input file. Any data between the comment string and the end of the current line is ignored.

skipfooter [int, default 0] Rows at the end to skip (0-indexed).

convert_float [bool, default True] Convert integral floats to int (i.e., 1.0 –> 1). If False, all numeric data will be read in as floats: Excel stores all numbers as floats internally.

mangle_dupe_cols [bool, default True] Duplicate columns will be specified as 'X', 'X.1', …'X.N', rather than 'X'…'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

****kwds** [optional] Optional keyword arguments can be passed to `TextFileReader`.

Returns

DataFrame or dict of DataFrames DataFrame from the passed in Excel file. See notes in sheet_name argument for more information on when a dict of DataFrames is returned.

See also:

`to_excel` Write DataFrame to an Excel file.

`to_csv` Write DataFrame to a comma-separated values (csv) file.

`read_csv` Read a comma-separated values (csv) file into DataFrame.

`read_fwf` Read a table of fixed-width formatted lines into DataFrame.

Examples

The file can be read using the file name as string or an open file object:

```
>>> pd.read_excel('tmp.xlsx', index_col=0)
       Name  Value
0   string1      1
1   string2      2
2  #Comment      3
```

```
>>> pd.read_excel(open('tmp.xlsx', 'rb'),
...               sheet_name='Sheet3')
   Unnamed: 0      Name  Value
0           0   string1      1
1           1   string2      2
2           2  #Comment      3
```

Index and header can be specified via the *index_col* and *header* arguments

```
>>> pd.read_excel('tmp.xlsx', index_col=None, header=None)
     0         1      2
0  NaN      Name  Value
1  0.0   string1      1
2  1.0   string2      2
3  2.0  #Comment      3
```

Column types are inferred but can be explicitly specified

```
>>> pd.read_excel('tmp.xlsx', index_col=0,
...               dtype={'Name': str, 'Value': float})
       Name  Value
0   string1    1.0
1   string2    2.0
2  #Comment    3.0
```

True, False, and NA values, and thousands separators have defaults, but can be explicitly specified, too. Supply the values you would like as strings or lists of strings!

```
>>> pd.read_excel('tmp.xlsx', index_col=0,
...               na_values=['string1', 'string2'])
       Name  Value
0       NaN      1
1       NaN      2
2  #Comment      3
```

Comment lines in the excel input file can be skipped using the *comment* kwarg

```
>>> pd.read_excel('tmp.xlsx', index_col=0, comment='#')
      Name  Value
0  string1    1.0
1  string2    2.0
2     None    NaN
```

pandas.ExcelFile.parse

`ExcelFile.`**`parse`**(*self*, *sheet_name=0*, *header=0*, *names=None*, *index_col=None*, *usecols=None*, *squeeze=False*, *converters=None*, *true_values=None*, *false_values=None*, *skiprows=None*, *nrows=None*, *na_values=None*, *parse_dates=False*, *date_parser=None*, *thousands=None*, *comment=None*, *skipfooter=0*, *convert_float=True*, *mangle_dupe_cols=True*, ***kwds*)

Parse specified sheet(s) into a DataFrame.

Equivalent to read_excel(ExcelFile, . . .) See the read_excel docstring for more info on accepted parameters.

Returns

DataFrame or dict of DataFrames DataFrame from the passed in Excel file.

`ExcelWriter`(path[, engine])	Class for writing DataFrame objects into excel sheets.

pandas.ExcelWriter

`class` `pandas.`**`ExcelWriter`**(*path*, *engine=None*, ***kwargs*)

Class for writing DataFrame objects into excel sheets.

Default is to use xlwt for xls, openpyxl for xlsx. See DataFrame.to_excel for typical usage.

Parameters

path [str] Path to xls or xlsx file.

engine [str (optional)] Engine to use for writing. If None, defaults to `io.excel.<extension>.writer`. NOTE: can only be passed as a keyword argument.

date_format [str, default None] Format string for dates written into Excel files (e.g. 'YYYY-MM-DD').

datetime_format [str, default None] Format string for datetime objects written into Excel files. (e.g. 'YYYY-MM-DD HH:MM:SS').

mode [{'w', 'a'}, default 'w'] File mode to use (write or append).

New in version 0.24.0.

Notes

None of the methods and properties are considered public.

For compatibility with CSV writers, ExcelWriter serializes lists and dicts to strings before writing.

Examples

Default usage:

```
>>> with ExcelWriter('path_to_file.xlsx') as writer:
...     df.to_excel(writer)
```

To write to separate sheets in a single file:

```
>>> with ExcelWriter('path_to_file.xlsx') as writer:
...     df1.to_excel(writer, sheet_name='Sheet1')
...     df2.to_excel(writer, sheet_name='Sheet2')
```

You can set the date format or datetime format:

```
>>> with ExcelWriter('path_to_file.xlsx',
                        date_format='YYYY-MM-DD',
                        datetime_format='YYYY-MM-DD HH:MM:SS') as writer:
...     df.to_excel(writer)
```

You can also append to an existing Excel file:

```
>>> with ExcelWriter('path_to_file.xlsx', mode='a') as writer:
...     df.to_excel(writer, sheet_name='Sheet3')
```

Attributes

None	

Methods

None	

3.1.5 JSON

read_json([path_or_buf, orient, typ, dtype, ...])	Convert a JSON string to pandas object.
json_normalize(data, List[Dict]], ...)	Normalize semi-structured JSON data into a flat table.

pandas.read_json

`pandas.`**`read_json`**`(`*path_or_buf=None*, *orient=None*, *typ='frame'*, *dtype=None*, *convert_axes=None*, *convert_dates=True*, *keep_default_dates=True*, *numpy=False*, *precise_float=False*, *date_unit=None*, *encoding=None*, *lines=False*, *chunksize=None*, *compression='infer'*`)`

Convert a JSON string to pandas object.

Parameters

path_or_buf [a valid JSON str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.json`.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

orient [str] Indication of expected JSON string format. Compatible JSON strings can be produced by `to_json()` with a corresponding orient value. The set of possible orients is:

- `'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

The allowed and default values depend on the value of the *typ* parameter.

- when `typ == 'series'`,
  - allowed orients are `{'split','records','index'}`
  - default is `'index'`
  - The Series index must be unique for orient `'index'`.
- when `typ == 'frame'`,
  - allowed orients are `{'split','records','index', 'columns', 'values', 'table'}`

  - default is `'columns'`
  - The DataFrame index must be unique for orients `'index'` and `'columns'`.
  - The DataFrame columns must be unique for orients `'index'`, `'columns'`, and `'records'`.

  New in version 0.23.0: 'table' as an allowed value for the `orient` argument

typy [{'frame', 'series'}, default 'frame'] The type of object to recover.

dtype [bool or dict, default None] If True, infer dtypes; if a dict of column to dtype, then use those; if False, then don't infer dtypes at all, applies only to the data.

For all `orient` values except `'table'`, default is True.

Changed in version 0.25.0: Not applicable for `orient='table'`.

convert_axes [bool, default None] Try to convert the axes to the proper dtypes.

For all `orient` values except `'table'`, default is True.

Changed in version 0.25.0: Not applicable for `orient='table'`.

convert_dates [bool or list of str, default True] List of columns to parse for dates. If True, then try to parse datelike columns. A column label is datelike if

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

keep_default_dates [bool, default True] If parsing dates, then parse the default datelike columns.

numpy [bool, default False] Direct decoding to numpy arrays. Supports numeric data only, but non-numeric column and index labels are supported. Note also that the JSON ordering MUST be the same for each term if numpy=True.

Deprecated since version 1.0.0.

precise_float [bool, 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 [str, default None] The timestamp unit to detect if converting dates. The default behaviour is to try and detect the correct precision, but if this is not desired then pass one of 's', 'ms', 'us' or 'ns' to force parsing only seconds, milliseconds, microseconds or nanoseconds respectively.

encoding [str, default is 'utf-8'] The encoding to use to decode py3 bytes.

lines [bool, default False] Read the file as a json object per line.

chunksize [int, optional] Return JsonReader object for iteration. See the line-delimited json docs for more information on `chunksize`. This can only be passed if *lines=True*. If this is None, the file will be read into memory all at once.

New in version 0.21.0.

compression [{'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'] For on-the-fly decompression of on-disk data. If 'infer', then use gzip, bz2, zip or xz if path_or_buf is a string ending in '.gz', '.bz2', '.zip', or 'xz', respectively, and no decompression otherwise. If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.21.0.

Returns

Series or DataFrame The type returned depends on the value of *typ*.

See also:

`DataFrame.to_json` Convert a DataFrame to a JSON string.

`Series.to_json` Convert a Series to a JSON string.

Notes

Specific to `orient='table'`, if a `DataFrame` with a literal `Index` name of *index* gets written with `to_json()`, the subsequent read operation will incorrectly set the `Index` name to `None`. This is because *index* is also used by `DataFrame.to_json()` to denote a missing `Index` name, and the subsequent `read_json()` operation cannot distinguish between the two. The same limitation is encountered with a `MultiIndex` and any names beginning with `'level_'`.

Examples

```
>>> df = pd.DataFrame([['a', 'b'], ['c', 'd']],
...                   index=['row 1', 'row 2'],
...                   columns=['col 1', 'col 2'])
```

Encoding/decoding a Dataframe using `'split'` formatted JSON:

```
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]}'
>>> pd.read_json(_, orient='split')
      col 1 col 2
row 1     a     b
row 2     c     d
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
>>> pd.read_json(_, orient='index')
      col 1 col 2
row 1     a     b
row 2     c     d
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"},{"col 1":"c","col 2":"d"}]'
>>> pd.read_json(_, orient='records')
  col 1 col 2
0     a     b
1     c     d
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
                "primaryKey": "index",
                "pandas_version": "0.20.0"},
    "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
            {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

pandas.json_normalize

pandas.**json_normalize**(*data: Union[Dict, List[Dict]], record_path: Union[str, List, NoneType] = None, meta: Union[str, List[Union[str, List[str]]], NoneType] = None, meta_prefix: Union[str, NoneType] = None, record_prefix: Union[str, NoneType] = None, errors: Union[str, NoneType] = 'raise', sep: str = '.', max_level: Union[int, NoneType] = None*) → 'DataFrame'

Normalize semi-structured JSON data into a flat table.

Parameters

data [dict or list of dicts] Unserialized JSON objects.

record_path [str or list of str, default None] Path in each object to list of records. If not passed, data will be assumed to be an array of records.

meta [list of paths (str or list of str), default None] Fields to use as metadata for each record in resulting table.

meta_prefix [str, default None] If True, prefix records with dotted (?) path, e.g. foo.bar.field if meta is [‘foo’, ‘bar’].

record_prefix [str, default None] If True, prefix records with dotted (?) path, e.g. foo.bar.field if path to records is [‘foo’, ‘bar’].

errors [{‘raise’, ‘ignore’}, default ‘raise’] Configures error handling.

- ‘ignore’ : will ignore KeyError if keys listed in meta are not always present.
- ‘raise’ : will raise KeyError if keys listed in meta are not always present.

sep [str, default ‘.’] Nested records will generate names separated by sep. e.g., for sep=’.’, {‘foo’: {‘bar’: 0}} -> foo.bar.

max_level [int, default None] Max number of levels(depth of dict) to normalize. if None, normalizes all levels.

New in version 0.25.0.

Returns

frame [DataFrame]

Normalize semi-structured JSON data into a flat table.

Examples

```
>>> from pandas.io.json import json_normalize
>>> data = [{'id': 1, 'name': {'first': 'Coleen', 'last': 'Volk'}},
...         {'name': {'given': 'Mose', 'family': 'Regner'}},
...         {'id': 2, 'name': 'Faye Raker'}]
>>> json_normalize(data)
    id        name name.family name.first name.given name.last
0  1.0         NaN         NaN     Coleen        NaN      Volk
1  NaN         NaN      Regner        NaN       Mose       NaN
2  2.0  Faye Raker         NaN        NaN        NaN       NaN
```

```
>>> data = [{'id': 1,
...          'name': "Cole Volk",
...          'fitness': {'height': 130, 'weight': 60}},
...         {'name': "Mose Reg",
...          'fitness': {'height': 130, 'weight': 60}},
...         {'id': 2, 'name': 'Faye Raker',
...          'fitness': {'height': 130, 'weight': 60}}]
>>> json_normalize(data, max_level=0)
            fitness                 id        name
0   {'height': 130, 'weight': 60}  1.0   Cole Volk
1   {'height': 130, 'weight': 60}  NaN    Mose Reg
2   {'height': 130, 'weight': 60}  2.0  Faye Raker
```

Normalizes nested data up to level 1.

```
>>> data = [{'id': 1,
...          'name': "Cole Volk",
...          'fitness': {'height': 130, 'weight': 60}},
...         {'name': "Mose Reg",
...          'fitness': {'height': 130, 'weight': 60}},
...         {'id': 2, 'name': 'Faye Raker',
...          'fitness': {'height': 130, 'weight': 60}}]
>>> json_normalize(data, max_level=1)
  fitness.height  fitness.weight   id    name
0   130              60          1.0    Cole Volk
1   130              60          NaN    Mose Reg
2   130              60          2.0    Faye Raker
```

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

(continues on next page)

(continued from previous page)

```
        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
```

```
>>> data = {'A': [1, 2]}
>>> json_normalize(data, 'A', record_prefix='Prefix.')
    Prefix.0
0          1
1          2
```

Returns normalized data with columns prefixed with the given string.

build_table_schema(data[, index, ...])	Create a Table schema from `data`.

pandas.io.json.build_table_schema

`pandas.io.json.`**`build_table_schema`**(*data*, *index=True*, *primary_key=None*, *version=True*)

Create a Table schema from `data`.

Parameters

data [Series, DataFrame]

index [bool, default True] Whether to include `data.index` in the schema.

primary_key [bool or None, default True] Column names to designate as the primary key. The default *None* will set *'primaryKey'* to the index level or levels if the index is unique.

version [bool, default True] Whether to include a field *pandas_version* with the version of pandas that generated the schema.

Returns

schema [dict]

Notes

See *_as_json_table_type* for conversion types. Timedeltas as converted to ISO8601 duration format with 9 decimal places after the seconds field for nanosecond precision.

Categoricals are converted to the *any* dtype, and use the *enum* field constraint to list the allowed values. The *ordered* attribute is included in an *ordered* field.

Examples

```
>>> 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'))
>>> build_table_schema(df)
{'fields': [{'name': 'idx', 'type': 'integer'},
{'name': 'A', 'type': 'integer'},
{'name': 'B', 'type': 'string'},
{'name': 'C', 'type': 'datetime'}],
'pandas_version': '0.20.0',
'primaryKey': ['idx']}
```

3.1.6 HTML

read_html(io[, match, flavor, header, . . .])	Read HTML tables into a `list` of `DataFrame` objects.

pandas.read_html

pandas.**read_html**(*io*, *match='.+'*, *flavor=None*, *header=None*, *index_col=None*, *skiprows=None*, *attrs=None*, *parse_dates=False*, *thousands=','*, *encoding=None*, *decimal='.'*, *converters=None*, *na_values=None*, *keep_default_na=True*, *displayed_only=True*)

Read HTML tables into a `list` of `DataFrame` objects.

Parameters

io [str, path object or file-like object] A URL, a file-like object, or a raw string containing HTML. Note that lxml only accepts the http, ftp and file url protocols. If you have a URL that starts with `'https'` you might try removing the `'s'`.

match [str or compiled regular expression, optional] The set of tables containing text matching this regex or string will be returned. Unless the HTML is extremely simple you will probably need to pass a non-empty string here. Defaults to '.+' (match any non-empty string). The default value will return all tables contained on a page. This value is converted to a regular expression so that there is consistent behavior between Beautiful Soup and lxml.

flavor [str or None] The parsing engine to use. 'bs4' and 'html5lib' are synonymous with each other, they are both there for backwards compatibility. The default of `None` tries to use `lxml` to parse and if that fails it falls back on `bs4` + `html5lib`.

header [int or list-like or None, optional] The row (or list of rows for a *MultiIndex*) to use to make the columns headers.

index_col [int or list-like or None, optional] The column (or list of columns) to use to create the index.

skiprows [int or list-like or slice or None, optional] Number of rows to skip after parsing the column integer. 0-based. If a sequence of integers or a slice is given, will skip the rows indexed by that sequence. Note that a single element sequence means 'skip the nth row' whereas an integer means 'skip n rows'.

attrs [dict or None, optional] This is a dictionary of attributes that you can pass to use to identify the table in the HTML. These are not checked for validity before being passed to lxml or Beautiful Soup. However, these attributes must be valid HTML table attributes to work correctly. For example,

```
attrs = {'id': 'table'}
```

is a valid attribute dictionary because the 'id' HTML tag attribute is a valid HTML attribute for *any* HTML tag as per this document.

```
attrs = {'asdf': 'table'}
```

is *not* a valid attribute dictionary because 'asdf' is not a valid HTML attribute even if it is a valid XML attribute. Valid HTML 4.01 table attributes can be found here. A working draft of the HTML 5 spec can be found here. It contains the latest information on table attributes for the modern web.

parse_dates [bool, optional] See `read_csv()` for more details.

thousands [str, optional] Separator to use to parse thousands. Defaults to `','`.

encoding [str or None, optional] The encoding used to decode the web page. Defaults to `None`.``None`` preserves the previous encoding behavior, which depends on the underlying parser library (e.g., the parser library will try to use the encoding provided by the document).

decimal [str, default '.'] Character to recognize as decimal point (e.g. use ',' for European data).

converters [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels, values are functions that take one input argument, the cell (not column) content, and return the transformed content.

na_values [iterable, default None] Custom NA values.

keep_default_na [bool, default True] If na_values are specified and keep_default_na is False the default NaN values are overridden, otherwise they're appended to.

displayed_only [bool, default True] Whether elements with "display: none" should be parsed.

Returns

dfs A list of DataFrames.

See also:

`read_csv`

Notes

Before using this function you should read the *gotchas about the HTML parsing libraries*.

Expect to do some cleanup after you call this function. For example, you might need to manually assign column names if the column names are converted to NaN when you pass the *header=0* argument. We try to assume as little as possible about the structure of the table and push the idiosyncrasies of the HTML contained in the table to the user.

This function searches for `<table>` elements and only for `<tr>` and `<th>` rows and `<td>` elements within each `<tr>` or `<th>` element in the table. `<td>` stands for "table data". This function attempts to properly handle `colspan` and `rowspan` attributes. If the function has a `<thead>` argument, it is used to construct the header, otherwise the function attempts to find the header within the body (by putting rows with only `<th>` elements into the header).

New in version 0.21.0.

Similar to `read_csv()` the *header* argument is applied **after** *skiprows* is applied.

This function will *always* return a list of `DataFrame` *or* it will fail, e.g., it will *not* return an empty list.

Examples

See the *read_html documentation in the IO section of the docs* for some examples of reading in HTML tables.

3.1.7 HDFStore: PyTables (HDF5)

`read_hdf`(path_or_buf[, key, where, columns, …])	Read from the store, close it if we opened it.
`HDFStore.put`(self, key, value[, format, …])	Store object in HDFStore.
`HDFStore.append`(self, key, value[, format, …])	Append to Table in file.
`HDFStore.get`(self, key)	Retrieve pandas object stored in file.
`HDFStore.select`(self, key[, where, start, …])	Retrieve pandas object stored in file, optionally based on where criteria.
`HDFStore.info`(self)	Print detailed information on the store.
`HDFStore.keys`(self)	Return a list of keys corresponding to objects stored in HDFStore.
`HDFStore.groups`(self)	Return a list of all the top-level nodes.
`HDFStore.walk`(self[, where])	Walk the pytables group hierarchy for pandas objects.

pandas.read_hdf

`pandas.`**`read_hdf`**(*path_or_buf*, *key=None*, *mode: str = 'r'*, *errors: str = 'strict'*, *where=None*, *start: Union[int, NoneType] = None*, *stop: Union[int, NoneType] = None*, *columns=None*, *iterator=False*, *chunksize: Union[int, NoneType] = None*, ***kwargs*)

Read from the store, close it if we opened it.

Retrieve pandas object stored in file, optionally based on where criteria

Parameters

path_or_buf [str, path object, pandas.HDFStore or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.h5`.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

Alternatively, pandas accepts an open `pandas.HDFStore` object.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

New in version 0.21.0: support for __fspath__ protocol.

key [object, optional] The group identifier in the store. Can be omitted if the HDF file contains a single pandas object.

mode [{'r', 'r+', 'a'}, default 'r'] Mode to use when opening the file. Ignored if path_or_buf is a `pandas.HDFStore`. Default is 'r'.

where [list, optional] A list of Term (or convertible) objects.

start [int, optional] Row number to start selection.

stop [int, optional] Row number to stop selection.

columns [list, optional] A list of columns names to return.

iterator [bool, optional] Return an iterator object.

chunksize [int, optional] Number of rows to include in an iteration when using an iterator.

errors [str, default 'strict'] Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

****kwargs** Additional keyword arguments passed to HDFStore.

Returns

item [object] The selected object. Return type depends on the object stored.

See also:

`DataFrame.to_hdf` Write a HDF file from a DataFrame.

`HDFStore` Low-level access to HDF files.

Examples

```
>>> df = pd.DataFrame([[1, 1.0, 'a']], columns=['x', 'y', 'z'])
>>> df.to_hdf('./store.h5', 'data')
>>> reread = pd.read_hdf('./store.h5')
```

pandas.HDFStore.put

`HDFStore.put`(*self*, *key: str*, *value: ~ FrameOrSeries*, *format=None*, *index=True*, *append=False*, *complib=None*, *complevel: Union[int, NoneType] = None*, *min_itemsize: Union[int, Dict[str, int], NoneType] = None*, *nan_rep=None*, *data_columns: Union[List[str], NoneType] = None*, *encoding=None*, *errors: str = 'strict'*)

Store object in HDFStore.

Parameters

key [str]

value [{Series, DataFrame}]

format ['fixed(f)|table(t)', default is 'fixed']

fixed(f) [Fixed format] Fast writing/reading. Not-appendable, nor searchable.

table(t) [Table format] Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default False] This will force Table format, append the input data to the existing.

data_columns [list, default None] List of columns to create as data columns, or True to use all columns. See here.

encoding [str, default None] Provide an encoding for strings.

dropna [bool, default False, do not write an ALL nan row to] The store settable by the option 'io.hdf.dropna_table'.

pandas.HDFStore.append

`HDFStore.append`(*self*, *key: str*, *value: ~ FrameOrSeries*, *format=None*, *axes=None*, *index=True*, *append=True*, *complib=None*, *complevel: Union[int, NoneType] = None*, *columns=None*, *min_itemsize: Union[int, Dict[str, int], NoneType] = None*, *nan_rep=None*, *chunksize=None*, *expectedrows=None*, *dropna: Union[bool, NoneType] = None*, *data_columns: Union[List[str], NoneType] = None*, *encoding=None*, *errors: str = 'strict'*)

Append to Table in file. Node must already exist and be Table format.

Parameters

key [str]

value [{Series, DataFrame}]

format ['table' is the default]

table(t) [table format] Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

append [bool, default True] Append the input data to the existing.

data_columns [list of columns, or True, default None] List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See here.

min_itemsize [dict of columns that specify minimum string sizes]

nan_rep [string to use as string nan representation]

chunksize [size to chunk the writing]

expectedrows [expected TOTAL row size of this table]

encoding [default None, provide an encoding for strings]

dropna [bool, default False] Do not write an ALL nan row to the store settable by the option 'io.hdf.dropna_table'.

Notes

Does *not* check if data being appended overlaps with existing data in the table, so be careful

pandas.HDFStore.get

`HDFStore.get`(*self*, *key: str*)

Retrieve pandas object stored in file.

Parameters

key [str]

Returns

object Same type as object stored in file.

pandas.HDFStore.select

`HDFStore.select`(*self*, *key: str*, *where=None*, *start=None*, *stop=None*, *columns=None*, *iterator=False*, *chunksize=None*, *auto_close: bool = False*)

Retrieve pandas object stored in file, optionally based on where criteria.

Parameters

key [str] Object being retrieved from file.

where [list, default None] List of Term (or convertible) objects, optional.

start [int, default None] Row number to start selection.

stop [int, default None] Row number to stop selection.

columns [list, default None] A list of columns that if not None, will limit the return columns.

iterator [bool, default False] Returns an iterator.

chunksize [int, default None] Number or rows to include in iteration, return an iterator.

auto_close [bool, default False] Should automatically close the store when finished.

Returns

object Retrieved object from file.

pandas.HDFStore.info

`HDFStore.info`(*self*) → str

Print detailed information on the store.

New in version 0.21.0.

Returns

str

pandas.HDFStore.keys

`HDFStore.keys`(*self*) → List[str]

Return a list of keys corresponding to objects stored in HDFStore.

Returns

list List of ABSOLUTE path-names (e.g. have the leading '/').

pandas.HDFStore.groups

`HDFStore.groups`(*self*)

Return a list of all the top-level nodes.

Each node returned is not a pandas storage object.

Returns

list List of objects.

pandas.HDFStore.walk

`HDFStore.walk`(*self*, *where='/'*)

Walk the pytables group hierarchy for pandas objects.

This generator will yield the group path, subgroups and pandas object names for each group.

Any non-pandas PyTables objects that are not a group will be ignored.

The *where* group itself is listed first (preorder), then each of its child groups (following an alphanumerical order) is also traversed, following the same procedure.

New in version 0.24.0.

Parameters

where [str, default "/"] Group where to start walking.

Yields

path [str] Full path to a group (without trailing '/').

groups [list] Names (strings) of the groups contained in *path*.

leaves [list] Names (strings) of the pandas objects contained in *path*.

3.1.8 Feather

read_feather(path[, columns])	Load a feather-format object from the file path.

pandas.read_feather

`pandas.read_feather`(*path*, *columns=None*, *use_threads: bool = True*)

Load a feather-format object from the file path.

Parameters

path [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.feather`.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

columns [sequence, default None] If not provided, all columns are read.

New in version 0.24.0.

use_threads [bool, default True]

Whether to parallelize reading using multiple threads.

New in version 0.24.0.

Returns

type of object stored in file

3.1.9 Parquet

`read_parquet`(path, engine[, columns])	Load a parquet object from the file path, returning a DataFrame.

pandas.read_parquet

pandas.**read_parquet**(*path*, *engine: str = 'auto'*, *columns=None*, ***kwargs*)

Load a parquet object from the file path, returning a DataFrame.

New in version 0.21.0.

Parameters

path [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.parquet`. A file URL can also be a path to a directory that contains multiple partitioned parquet files. Both pyarrow and fastparquet support paths to directories as well as file URLs. A directory path could be: `file://localhost/path/to/tables`

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

engine [{'auto', 'pyarrow', 'fastparquet'}, default 'auto'] Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

columns [list, default=None] If not None, only these columns will be read from the file.

New in version 0.21.1.

****kwargs** Any additional kwargs are passed to the engine.

Returns

DataFrame

3.1.10 ORC

`read_orc`(path, pathlib.Path, IO[~AnyStr]], …)	Load an ORC object from the file path, returning a DataFrame.

pandas.read_orc

pandas.**read_orc**(*path: Union[str, pathlib.Path, IO[~ AnyStr]]*, *columns: Union[List[str], NoneType] = None*, ***kwargs*) → 'DataFrame'

Load an ORC object from the file path, returning a DataFrame.

New in version 1.0.0.

Parameters

path [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.orc`.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

columns [list, default None] If not None, only these columns will be read from the file.

****kwargs** Any additional kwargs are passed to pyarrow.

Returns

DataFrame

3.1.11 SAS

`read_sas`(filepath_or_buffer[, format, . . .])	Read SAS files stored as either XPORT or SAS7BDAT format files.

pandas.read_sas

`pandas.`**`read_sas`**(*filepath_or_buffer*, *format=None*, *index=None*, *encoding=None*, *chunksize=None*, *iterator=False*)

Read SAS files stored as either XPORT or SAS7BDAT format files.

Parameters

filepath_or_buffer [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. A local file could be: `file://localhost/path/to/table.sas`.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handler (e.g. via builtin `open` function) or `StringIO`.

format [str {'xport', 'sas7bdat'} or None] If None, file format is inferred from file extension. If 'xport' or 'sas7bdat', uses the corresponding format.

index [identifier of index column, defaults to None] Identifier of column that should be used as index of the DataFrame.

encoding [str, default is None] Encoding for text data. If None, text data are stored as raw bytes.

chunksize [int] Read file *chunksize* lines at a time, returns iterator.

iterator [bool, defaults to False] If True, returns an iterator for reading the file incrementally.

Returns

DataFrame if iterator=False and chunksize=None, else SAS7BDATReader

or XportReader

3.1.12 SPSS

`read_spss`(path, pathlib.Path], usecols, . . .)	Load an SPSS file from the file path, returning a DataFrame.

pandas.read_spss

pandas.**read_spss**(*path: Union[str, pathlib.Path], usecols: Union[Sequence[str], NoneType] = None, convert_categoricals: bool = True*) → pandas.core.frame.DataFrame

Load an SPSS file from the file path, returning a DataFrame.

New in version 0.25.0.

Parameters

path [string or Path] File path.

usecols [list-like, optional] Return a subset of the columns. If None, return all columns.

convert_categoricals [bool, default is True] Convert categorical columns into pd.Categorical.

Returns

DataFrame

3.1.13 SQL

`read_sql_table`(table_name, con[, schema, . . .])	Read SQL database table into a DataFrame.
`read_sql_query`(sql, con[, index_col, . . .])	Read SQL query into a DataFrame.
`read_sql`(sql, con[, index_col, . . .])	Read SQL query or database table into a DataFrame.

pandas.read_sql_table

pandas.**read_sql_table**(*table_name, con, schema=None, index_col=None, coerce_float=True, parse_dates=None, columns=None, chunksize=None*)

Read SQL database table into a DataFrame.

Given a table name and a SQLAlchemy connectable, returns a DataFrame. This function does not support DBAPI connections.

Parameters

table_name [str] Name of SQL table in database.

con [SQLAlchemy connectable or str] A database URI could be provided as as str. SQLite DBAPI connection mode not supported.

schema [str, default None] Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).

index_col [str or list of str, optional, default: None] Column(s) to set as index(MultiIndex).

coerce_float [bool, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Can result in loss of Precision.

parse_dates [list or dict, default None]

- List of column names to parse as dates.

- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of `pandas.to_datetime()` Especially useful with databases without native Datetime support, such as SQLite.

columns [list, default None] List of column names to select from SQL table.

chunksize [int, default None] If specified, returns an iterator where *chunksize* is the number of rows to include in each chunk.

Returns

DataFrame A SQL table is returned as two-dimensional data structure with labeled axes.

See also:

`read_sql_query` Read SQL query into a DataFrame.

`read_sql` Read SQL query or database table into a DataFrame.

Notes

Any datetime values with time zone information will be converted to UTC.

Examples

```
>>> pd.read_sql_table('table_name', 'postgres:///db_name')
```

pandas.read_sql_query

`pandas.`**`read_sql_query`**(*sql*, *con*, *index_col=None*, *coerce_float=True*, *params=None*, *parse_dates=None*, *chunksize=None*)

Read SQL query into a DataFrame.

Returns a DataFrame corresponding to the result set of the query string. Optionally provide an *index_col* parameter to use one of the columns as the index, otherwise default integer index will be used.

Parameters

sql [str SQL query or SQLAlchemy Selectable (select or text object)] SQL query to be executed.

con [SQLAlchemy connectable(engine/connection), database str URI,] or sqlite3 DBAPI2 connection Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

index_col [str or list of strings, optional, default: None] Column(s) to set as index(MultiIndex).

coerce_float [bool, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Useful for SQL result sets.

params [list, tuple or dict, optional, default: None] List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported. Eg. for psycopg2, uses %(name)s so use params={'name' : 'value'}.

parse_dates [list or dict, default: None]

- List of column names to parse as dates.
- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of *`pandas.to_datetime()`* Especially useful with databases without native Datetime support, such as SQLite.

chunksize [int, default None] If specified, return an iterator where *chunksize* is the number of rows to include in each chunk.

Returns

DataFrame

See also:

`read_sql_table` Read SQL database table into a DataFrame.

`read_sql`

Notes

Any datetime values with time zone information parsed via the *parse_dates* parameter will be converted to UTC.

pandas.read_sql

`pandas.`**`read_sql`**`(`*sql*, *con*, *index_col=None*, *coerce_float=True*, *params=None*, *parse_dates=None*, *columns=None*, *chunksize=None*`)`

Read SQL query or database table into a DataFrame.

This function is a convenience wrapper around `read_sql_table` and `read_sql_query` (for backward compatibility). It will delegate to the specific function depending on the provided input. A SQL query will be routed to `read_sql_query`, while a database table name will be routed to `read_sql_table`. Note that the delegated function might have more specific notes about their functionality not listed here.

Parameters

sql [str or SQLAlchemy Selectable (select or text object)] SQL query to be executed or a table name.

con [SQLAlchemy connectable (engine/connection) or database str URI] or DBAPI2 connection (fallback mode)'

Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported. The user is responsible for engine disposal and connection closure for the SQLAlchemy connectable. See here

index_col [str or list of strings, optional, default: None] Column(s) to set as index(MultiIndex).

coerce_float [bool, default True] Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.

params [list, tuple or dict, optional, default: None] List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported. Eg. for psycopg2, uses %(name)s so use params={'name' : 'value'}.

parse_dates [list or dict, default: None]

- List of column names to parse as dates.
- Dict of `{column_name: format string}` where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of `{column_name: arg dict}`, where the arg dict corresponds to the keyword arguments of `pandas.to_datetime()` Especially useful with databases without native Datetime support, such as SQLite.

columns [list, default: None] List of column names to select from SQL table (only used when reading a table).

chunksize [int, default None] If specified, return an iterator where *chunksize* is the number of rows to include in each chunk.

Returns

DataFrame

See also:

`read_sql_table`	Read SQL database table into a DataFrame.
`read_sql_query`	Read SQL query into a DataFrame.

3.1.14 Google BigQuery

`read_gbq`(query, project_id, …[, …])	Load data from Google BigQuery.

pandas.read_gbq

`pandas.`**`read_gbq`**(*query: str, project_id: Union[str, NoneType] = None, index_col: Union[str, NoneType] = None, col_order: Union[List[str], NoneType] = None, reauth: bool = False, auth_local_webserver: bool = False, dialect: Union[str, NoneType] = None, location: Union[str, NoneType] = None, configuration: Union[Dict[str, Any], NoneType] = None, credentials=None, use_bqstorage_api: Union[bool, NoneType] = None, private_key=None, verbose=None, progress_bar_type: Union[str, NoneType] = None*) → 'DataFrame'

Load data from Google BigQuery.

This function requires the pandas-gbq package.

See the How to authenticate with Google BigQuery guide for authentication instructions.

Parameters

query [str] SQL-Like Query to return data values.

project_id [str, optional] Google BigQuery Account project ID. Optional when available from the environment.

index_col [str, optional] Name of result column to use for index in results DataFrame.

col_order [list(str), optional] List of BigQuery column names in the desired order for results DataFrame.

reauth [bool, default False] Force Google BigQuery to re-authenticate the user. This is useful if multiple accounts are used.

auth_local_webserver [bool, default False] Use the local webserver flow instead of the console flow when getting user credentials.

New in version 0.2.0 of pandas-gbq.

dialect [str, default 'legacy'] Note: The default value is changing to 'standard' in a future version.

SQL syntax dialect to use. Value can be one of:

`'legacy'` Use BigQuery's legacy SQL dialect. For more information see BigQuery Legacy SQL Reference.

`'standard'` Use BigQuery's standard SQL, which is compliant with the SQL 2011 standard. For more information see BigQuery Standard SQL Reference.

Changed in version 0.24.0.

location [str, optional] Location where the query job should run. See the BigQuery locations documentation for a list of available locations. The location must match that of any datasets used in the query.

New in version 0.5.0 of pandas-gbq.

configuration [dict, optional] Query config parameters for job processing. For example:

configuration = {'query': {'useQueryCache': False}}

For more information see BigQuery REST API Reference.

credentials [google.auth.credentials.Credentials, optional] Credentials for accessing Google APIs. Use this parameter to override default credentials, such as to use Compute Engine `google.auth.compute_engine.Credentials` or Service Account `google.oauth2.service_account.Credentials` directly.

New in version 0.8.0 of pandas-gbq.

New in version 0.24.0.

use_bqstorage_api [bool, default False] Use the BigQuery Storage API to download query results quickly, but at an increased cost. To use this API, first enable it in the Cloud Console. You must also have the bigquery.readsessions.create permission on the project you are billing queries to.

This feature requires version 0.10.0 or later of the `pandas-gbq` package. It also requires the `google-cloud-bigquery-storage` and `fastavro` packages.

New in version 0.25.0.

progress_bar_type [Optional, str] If set, use the tqdm library to display a progress bar while the data downloads. Install the `tqdm` package to use this feature.

Possible values of `progress_bar_type` include:

`None` No progress bar.

`'tqdm'` Use the `tqdm.tqdm()` function to print a progress bar to `sys.stderr`.

`'tqdm_notebook'` Use the `tqdm.tqdm_notebook()` function to display a progress bar as a Jupyter notebook widget.