IO tools (text, CSV, HDF5, ...)

The pandas I/O API is a set of top level `reader` functions accessed like `pandas.read_csv()` that generally return a pandas object. The corresponding `writer` functions are object methods that are accessed like `DataFrame.to_csv()`. Below is a table containing available `readers` and `writers`.

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	Fixed-Width Text File	read_fwf	
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	LaTeX		Styler.to_latex
text	XML	read_xml	to_xml
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	ORC Format	read_orc	to_orc
binary	Stata	read_stata	to_stata

Back to top

Skip to main content

Format Type	Data Description	Reader	Writer
binary	SPSS	read_spss	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google BigQuery	read_gbq	to_gbq

Here is an informal performance comparison for some of these IO methods.

Note

For examples that use the `StringIO` class, make sure you import it with `from io import StringIO` for Python 3.

CSV & text files

The workhorse function for reading text files (a.k.a. flat files) is `read_csv()`. See the cookbook for some advanced strategies.

Parsing options

`read_csv()` accepts the following common arguments:

Basic

filepath_or_buffer : *various*

Either a path to a file (a `str`, `pathlib.Path`, or `py:py._path.local.LocalPath`), URL (including http, ftp, and S3 locations), or any object with a `read()` method (such as an open file or `StringIO`).

sep : *str, defaults to* `','` *for* `read_csv()`, `\t` *for* `read_table()`

Delimiter to use. If sep is `None`, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1

Skip to main content

force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\\r\\t'`.

delimiter : *str, default* `None`

Alternative argument name for sep.

delim_whitespace : *boolean, default False*

Specifies whether or not whitespace (e.g. `' '` or `'\t'`) will be used as the delimiter. Equivalent to setting `sep='\s+'`. If this option is set to `True`, nothing should be passed in for the `delimiter` parameter.

Column and index locations and names

header : *int or list of ints, default* `'infer'`

Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names.

The header can be a list of ints that specify row locations for a MultiIndex on the columns e.g. `[0,1,3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so header=0 denotes the first line of data rather than the first line of the file.

names : *array-like, default* `None`

List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list are not allowed.

index_col : *int, str, sequence of int / str, or False, optional, default* `None`

Column(s) to use as the row labels of the `DataFrame`, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.

> **Note**
>
> `index_col=False` can be used to force pandas to *not* use the first column as the index, e.g. when you have a malformed file with delimiters at the end of each line.

Skip to main content

The default value of `None` instructs pandas to guess. If the number of fields in the column header row is equal to the number of fields in the body of the data file, then a default index is used. If it is larger, then the first columns are used as index so that the remaining number of fields in the body are equal to the number of fields in the header.

The first row after the header is used to determine the number of columns, which will go into the index. If the subsequent rows contain less columns than the first row, they are filled with `NaN`.

This can be avoided through `usecols`. This ensures that the columns are taken as is and the trailing data are ignored.

usecols : *list-like or callable, default* `None`

Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in `names` or inferred from the document header row(s). If `names` are given, the document header row(s) are not taken into account. For example, a valid list-like `usecols` parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`.

Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To instantiate a DataFrame from `data` with element order preserved use `pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]` for columns in `['foo', 'bar']` order or `pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]` for `['bar', 'foo']` order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True:

```
In [1]: import pandas as pd

In [2]: from io import StringIO

In [3]: data = "col1,col2,col3\na,b,1\na,b,2\nc,d,3"

In [4]: pd.read_csv(StringIO(data))
Out[4]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [5]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ["COL1", "COL3"]
Out[5]:
  col1  col3
0    a     1
1    a     2
2    c     3
```

Skip to main content

Using this parameter results in much faster parsing time and lower memory usage when using the c engine. The Python engine loads the data first before deciding which columns to drop.

General parsing configuration

dtype : *Type name or dict of column -> type, default* `None`

Data type for data or columns. E.g. `{'a': np.float64, 'b': np.int32, 'c': 'Int64'}` Use `str` or `object` together with suitable `na_values` settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

> ***New in version 1.5.0:*** Support for defaultdict was added. Specify a defaultdict as input where the default determines the dtype of the columns which are not explicitly listed.

dtype_backend : *{"numpy_nullable", "pyarrow"}, defaults to NumPy backed DataFrames*

Which dtype_backend to use, e.g. whether a DataFrame should have NumPy arrays, nullable dtypes are used for all dtypes that have a nullable implementation when "numpy_nullable" is set, pyarrow is used for all dtypes if "pyarrow" is set.

The dtype_backends are still experimental.

> ***New in version 2.0.***

engine : *{* `'c'`, `'python'`, `'pyarrow'` ***}***

Parser engine to use. The C and pyarrow engines are faster, while the python engine is currently more feature-complete. Multithreading is currently only supported by the pyarrow engine.

> ***New in version 1.4.0:*** The "pyarrow" engine was added as an *experimental* engine, and some features are unsupported, or may not work correctly, with this engine.

converters : *dict, default* `None`

Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

true_values : *list, default* `None`

Skip to main content

false_values : *list, default* `None`

Values to consider as `False`.

skipinitialspace : *boolean, default* `False`

Skip spaces after delimiter.

skiprows : *list-like or integer, default* `None`

Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise:

```
In [6]: data = "col1,col2,col3\na,b,1\na,b,2\nc,d,3"

In [7]: pd.read_csv(StringIO(data))
Out[7]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [8]: pd.read_csv(StringIO(data), skiprows=lambda x: x % 2 != 0)
Out[8]:
  col1 col2  col3
0    a    b     2
```

skipfooter : *int, default* `0`

Number of lines at bottom of file to skip (unsupported with engine='c').

nrows : *int, default* `None`

Number of rows of file to read. Useful for reading pieces of large files.

low_memory : *boolean, default* `True`

Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set `False`, or specify the type with the `dtype` parameter. Note that the entire file is read into a single `DataFrame` regardless, use the `chunksize` or `iterator` parameter to return the data in chunks. (Only valid with C parser)

memory_map : *boolean, default False*

If a filepath is provided for `filepath_or_buffer`, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

Skip to main content

NA and missing data handling

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. See na values const below for a list of the values interpreted as NaN by default.

keep_default_na : *boolean, 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 : *boolean, 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 : *boolean, default* `False`

Indicate number of NA values placed in non-numeric columns.

skip_blank_lines : *boolean, default* `True`

If `True`, skip over blank lines rather than interpreting as NaN values.

Datetime handling

parse_dates : *boolean or list of ints or names or list of lists or dict, default* `False`.

- If `True` -> try parsing the index.
- If `[1, 2, 3]` -> try parsing columns 1, 2, 3 each as a separate date column.
- If `[[1, 3]]` -> combine columns 1 and 3 and parse as a single date column.
- If `{'foo': [1, 3]}` -> parse columns 1, 3 as date and call result 'foo'

Skip to main content

> **Note**
>
> A fast-path exists for iso8601-formatted dates.

infer_datetime_format : *boolean, default* `False`

If `True` and parse_dates is enabled for a column, attempt to infer the datetime format to speed up the processing.

> ***Deprecated since version 2.0.0:*** A strict version of this argument is now the default, passing it has no effect.

keep_date_col : *boolean, default* `False`

If `True` and parse_dates specifies combining multiple columns then keep the original columns.

date_parser : *function, default* `None`

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.

> ***Deprecated since version 2.0.0:*** Use `date_format` instead, or read in as `object` and then apply `to_datetime()` as-needed.

date_format : *str or dict of column -> format, default* `None`

If used in conjunction with `parse_dates`, will parse dates according to this format. For anything more complex, please read in as `object` and then apply `to_datetime()` as-needed.

> ***New in version 2.0.0.***

dayfirst : *boolean, default* `False`

DD/MM format dates, international and European format.

Skip to main content

If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets.

Iteration

iterator : *boolean, default* `False`

Return `TextFileReader` object for iteration or getting chunks with `get_chunk()`.

chunksize : *int, default* `None`

Return `TextFileReader` object for iteration. See iterating and chunking below.

Quoting, compression, and file format

compression : *{`'infer'`, `'gzip'`, `'bz2'`, `'zip'`, `'xz'`, `'zstd'`, `None`, `dict`}, default `'infer'`*

For on-the-fly decompression of on-disk data. If 'infer', then use gzip, bz2, zip, xz, or zstandard if `filepath_or_buffer` is path-like ending in '.gz', '.bz2', '.zip', '.xz', '.zst', 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. Can also be a dict with key `'method'` set to one of {`'zip'`, `'gzip'`, `'bz2'`, `'zstd'`} and other key-value pairs are forwarded to `zipfile.ZipFile`, `gzip.GzipFile`, `bz2.BZ2File`, or `zstandard.ZstdDecompressor`. As an example, the following could be passed for faster compression and to create a reproducible gzip archive: `compression={'method': 'gzip', 'compresslevel': 1, 'mtime': 1}`.

> ***Changed in version 1.2.0:*** Previous versions forwarded dict entries for 'gzip' to `gzip.open`.

thousands : *str, default* `None`

Thousands separator.

decimal : *str, default* `'.'`

Character to recognize as decimal point. E.g. use `','` for European data.

float_precision : *string, default None*

Specifies which converter the C engine should use for floating-point values. The options are

Skip to main content

the round-trip converter.

lineterminator : *str (length 1), default* `None`

Character to break file into lines. Only valid with C parser.

quotechar : *str (length 1)*

The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

quoting : *int or* `csv.QUOTE_*` *instance, default* `0`

Control field quoting behavior per `csv.QUOTE_*` constants. Use one of `QUOTE_MINIMAL` (0), `QUOTE_ALL` (1), `QUOTE_NONNUMERIC` (2) or `QUOTE_NONE` (3).

doublequote : *boolean, default* `True`

When `quotechar` is specified and `quoting` is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive `quotechar` elements **inside** a field as a single `quotechar` element.

escapechar : *str (length 1), default* `None`

One-character string used to escape delimiter when quoting is `QUOTE_NONE`.

comment : *str, default* `None`

Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter `header` but not by `skiprows`. For example, if `comment='#'`, parsing '#empty\na,b,c\n1,2,3' with `header=0` will result in 'a,b,c' being treated as the header.

encoding : *str, default* `None`

Encoding to use for UTF when reading/writing (e.g. `'utf-8'`). List of Python standard encodings.

dialect : *str or* `csv.Dialect` *instance, default* `None`

If provided, this parameter will override values (default or not) for the following parameters: `delimiter`, `doublequote`, `escapechar`, `skipinitialspace`, `quotechar`, and `quoting`. If it is necessary to override values, a ParserWarning will be issued. See `csv.Dialect` documentation for more details.

Error handling

on_bad_lines : *('error', 'warn', 'skip'), default 'error'*

Skip to main content

Specifies what to do upon encountering a bad line (a line with too many fields). Allowed values are :

- ‘error’, raise an ParserError when a bad line is encountered.
- ‘warn’, print a warning when a bad line is encountered and skip that line.
- ‘skip’, skip bad lines without raising or warning when they are encountered.

> ***New in version 1.3.0.***

Specifying column data types

You can indicate the data type for the whole `DataFrame` or individual columns:

```
In [9]: import numpy as np

In [10]: data = "a,b,c,d\n1,2,3,4\n5,6,7,8\n9,10,11"

In [11]: print(data)
a,b,c,d
1,2,3,4
5,6,7,8
9,10,11

In [12]: df = pd.read_csv(StringIO(data), dtype=object)

In [13]: df
Out[13]:
   a   b   c    d
0  1   2   3    4
1  5   6   7    8
2  9  10  11  NaN

In [14]: df["a"][0]
Out[14]: '1'

In [15]: df = pd.read_csv(StringIO(data), dtype={"b": object, "c": np.float64, "d": "In

In [16]: df.dtypes
Out[16]:
a      int64
b     object
c    float64
d      Int64
dtype: object
```

Fortunately, pandas offers more than one way to ensure that your column(s) contain only one

Skip to main content

and here to learn more about `object` conversion in pandas.

For instance, you can use the `converters` argument of `read_csv()`:

```
In [17]: data = "col_1\n1\n2\n'A'\n4.22"

In [18]: df = pd.read_csv(StringIO(data), converters={"col_1": str})

In [19]: df
Out[19]:
  col_1
0     1
1     2
2   'A'
3  4.22

In [20]: df["col_1"].apply(type).value_counts()
Out[20]:
col_1
<class 'str'>    4
Name: count, dtype: int64
```

Or you can use the `to_numeric()` function to coerce the dtypes after reading in the data,

```
In [21]: df2 = pd.read_csv(StringIO(data))

In [22]: df2["col_1"] = pd.to_numeric(df2["col_1"], errors="coerce")

In [23]: df2
Out[23]:
   col_1
0   1.00
1   2.00
2    NaN
3   4.22

In [24]: df2["col_1"].apply(type).value_counts()
Out[24]:
col_1
<class 'float'>    4
Name: count, dtype: int64
```

which will convert all valid parsing to floats, leaving the invalid parsing as `NaN`.

Ultimately, how you deal with reading in columns containing mixed dtypes depends on your specific needs. In the case above, if you wanted to `NaN` out the data anomalies, then `to_numeric()` is probably your best option. However, if you wanted for all the data to be coerced, no matter the type, then using the `converters` argument of `read_csv()` would certainly be worth trying.

Skip to main content

Note

In some cases, reading in abnormal data with columns containing mixed dtypes will result in an inconsistent dataset. If you rely on pandas to infer the dtypes of your columns, the parsing engine will go and infer the dtypes for different chunks of the data, rather than the whole dataset at once. Consequently, you can end up with column(s) with mixed dtypes. For example,

```
In [25]: col_1 = list(range(500000)) + ["a", "b"] + list(range(500000))

In [26]: df = pd.DataFrame({"col_1": col_1})

In [27]: df.to_csv("foo.csv")

In [28]: mixed_df = pd.read_csv("foo.csv")

In [29]: mixed_df["col_1"].apply(type).value_counts()
Out[29]:
col_1
<class 'int'>    737858
<class 'str'>    262144
Name: count, dtype: int64

In [30]: mixed_df["col_1"].dtype
Out[30]: dtype('O')
```

will result with `mixed_df` containing an `int` dtype for certain chunks of the column, and `str` for others due to the mixed dtypes from the data that was read in. It is important to note that the overall column will be marked with a `dtype` of `object`, which is used for columns with mixed dtypes.

Setting `dtype_backend="numpy_nullable"` will result in nullable dtypes for every column.

```
In [31]: data = """a,b,c,d,e,f,g,h,i,j
   ....: 1,2.5,True,a,,,,,12-31-2019,
   ....: 3,4.5,False,b,6,7.5,True,a,12-31-2019,
   ....: """
   ....:

In [32]: df = pd.read_csv(StringIO(data), dtype_backend="numpy_nullable", parse_dates=[

In [33]: df
Out[33]:
   a    b      c  d     e     f     g     h          i     j
0  1  2.5   True  a  <NA>  <NA>  <NA>  <NA> 2019-12-31  <NA>
1  3  4.5  False  b     6   7.5  True     a 2019-12-31  <NA>

In [34]: df.dtypes
```

Skip to main content

```
b           Float64
c           boolean
d    string[python]
e             Int64
f           Float64
g           boolean
h    string[python]
i    datetime64[ns]
j             Int64
dtype: object
```

Specifying categorical dtype

`Categorical` columns can be parsed directly by specifying `dtype='category'` or `dtype=CategoricalDtype(categories, ordered)`.

```
In [35]: data = "col1,col2,col3\na,b,1\a,b,2\nc,d,3"

In [36]: pd.read_csv(StringIO(data))
Out[36]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [37]: pd.read_csv(StringIO(data)).dtypes
Out[37]:
col1    object
col2    object
col3     int64
dtype: object

In [38]: pd.read_csv(StringIO(data), dtype="category").dtypes
Out[38]:
col1    category
col2    category
col3    category
dtype: object
```

Individual columns can be parsed as a `Categorical` using a dict specification:

```
In [39]: pd.read_csv(StringIO(data), dtype={"col1": "category"}).dtypes
Out[39]:
col1    category
col2      object
col3       int64
dtype: object
```

Skip to main content

Specifying `dtype='category'` will result in an unordered `Categorical` whose `categories` are the unique values observed in the data. For more control on the categories and order, create a `CategoricalDtype` ahead of time, and pass that for that column's `dtype`.

```
In [40]: from pandas.api.types import CategoricalDtype

In [41]: dtype = CategoricalDtype(["d", "c", "b", "a"], ordered=True)

In [42]: pd.read_csv(StringIO(data), dtype={"col1": dtype}).dtypes
Out[42]:
col1    category
col2      object
col3       int64
dtype: object
```

When using `dtype=CategoricalDtype`, "unexpected" values outside of `dtype.categories` are treated as missing values.

```
In [43]: dtype = CategoricalDtype(["a", "b", "d"])  # No 'c'

In [44]: pd.read_csv(StringIO(data), dtype={"col1": dtype}).col1
Out[44]:
0      a
1      a
2    NaN
Name: col1, dtype: category
Categories (3, object): ['a', 'b', 'd']
```

This matches the behavior of `Categorical.set_categories()`.

Skip to main content

Note

With `dtype='category'`, the resulting categories will always be parsed as strings (object dtype). If the categories are numeric they can be converted using the `to_numeric()` function, or as appropriate, another converter such as `to_datetime()`.

When `dtype` is a `CategoricalDtype` with homogeneous `categories` (all numeric, all datetimes, etc.), the conversion is done automatically.

```
In [45]: df = pd.read_csv(StringIO(data), dtype="category")

In [46]: df.dtypes
Out[46]:
col1    category
col2    category
col3    category
dtype: object

In [47]: df["col3"]
Out[47]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, object): ['1', '2', '3']

In [48]: new_categories = pd.to_numeric(df["col3"].cat.categories)

In [49]: df["col3"] = df["col3"].cat.rename_categories(new_categories)

In [50]: df["col3"]
Out[50]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, int64): [1, 2, 3]
```

Naming and using columns

Handling column names

A file may or may not have a header row. pandas assumes the first row should be used as the column names:

Skip to main content

```
In [52]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [53]: pd.read_csv(StringIO(data))
Out[53]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

By specifying the `names` argument in conjunction with `header` you can indicate other names to use and whether or not to throw away the header row (if any):

```
In [54]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [55]: pd.read_csv(StringIO(data), names=["foo", "bar", "baz"], header=0)
Out[55]:
   foo  bar  baz
0    1    2    3
1    4    5    6
2    7    8    9

In [56]: pd.read_csv(StringIO(data), names=["foo", "bar", "baz"], header=None)
Out[56]:
  foo bar baz
0   a   b   c
1   1   2   3
2   4   5   6
3   7   8   9
```

If the header is in a row other than the first, pass the row number to `header`. This will skip the preceding rows:

```
In [57]: data = "skip this skip it\na,b,c\n1,2,3\n4,5,6\n7,8,9"

In [58]: pd.read_csv(StringIO(data), header=1)
Out[58]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

Skip to main content

Note

Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first non-blank line of the file, if column names are passed explicitly then the behavior is identical to `header=None`.

Duplicate names parsing

If the file or header contains duplicate names, pandas will by default distinguish between them so as to prevent overwriting data:

```
In [59]: data = "a,b,a\n0,1,2\n3,4,5"

In [60]: pd.read_csv(StringIO(data))
Out[60]:
   a  b  a.1
0  0  1    2
1  3  4    5
```

There is no more duplicate data because duplicate columns 'X', ..., 'X' become 'X', 'X.1', ..., 'X.N'.

Filtering columns (`usecols`)

The `usecols` argument allows you to select any subset of the columns in a file, either using the column names, position numbers or a callable:

```
In [61]: data = "a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz"

In [62]: pd.read_csv(StringIO(data))
Out[62]:
   a  b  c    d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

In [63]: pd.read_csv(StringIO(data), usecols=["b", "d"])
Out[63]:
   b    d
0  2  foo
1  5  bar
2  8  baz

In [64]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
```

Skip to main content

```
0  1  3  foo
1  4  6  bar
2  7  9  baz

In [65]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ["A", "C"])
Out[65]:
   a  c
0  1  3
1  4  6
2  7  9
```

The `usecols` argument can also be used to specify which columns not to use in the final result:

```
In [66]: pd.read_csv(StringIO(data), usecols=lambda x: x not in ["a", "c"])
Out[66]:
   b    d
0  2  foo
1  5  bar
2  8  baz
```

In this case, the callable is specifying that we exclude the “a” and “c” columns from the output.

Comments and empty lines

Ignoring line comments and empty lines

If the `comment` parameter is specified, then completely commented lines will be ignored. By default, completely blank lines will be ignored as well.

```
In [67]: data = "\na,b,c\n  \n# commented line\n1,2,3\n\n4,5,6"

In [68]: print(data)

a,b,c

# commented line
1,2,3

4,5,6

In [69]: pd.read_csv(StringIO(data), comment="#")
Out[69]:
   a  b  c
0  1  2  3
1  4  5  6
```

Skip to main content

```
In [70]: data = "a,b,c\n\n1,2,3\n\n\n4,5,6"

In [71]: pd.read_csv(StringIO(data), skip_blank_lines=False)
Out[71]:
     a    b    c
0  NaN  NaN  NaN
1  1.0  2.0  3.0
2  NaN  NaN  NaN
3  NaN  NaN  NaN
4  4.0  5.0  6.0
```

Warning

The presence of ignored lines might create ambiguities involving line numbers; the parameter `header` uses row numbers (ignoring commented/empty lines), while `skiprows` uses line numbers (including commented/empty lines):

```
In [72]: data = "#comment\na,b,c\nA,B,C\n1,2,3"

In [73]: pd.read_csv(StringIO(data), comment="#", header=1)
Out[73]:
   A  B  C
0  1  2  3

In [74]: data = "A,B,C\n#comment\na,b,c\n1,2,3"

In [75]: pd.read_csv(StringIO(data), comment="#", skiprows=2)
Out[75]:
   a  b  c
0  1  2  3
```

If both `header` and `skiprows` are specified, `header` will be relative to the end of `skiprows`. For example:

```
In [76]: data = (
   ....:     "# empty\n"
   ....:     "# second empty line\n"
   ....:     "# third emptyline\n"
   ....:     "X,Y,Z\n"
   ....:     "1,2,3\n"
   ....:     "A,B,C\n"
   ....:     "1,2.,4.\n"
   ....:     "5.,NaN,10.0\n"
   ....: )
   ....:

In [77]: print(data)
# empty
```

Skip to main content

```
X,Y,Z
1,2,3
A,B,C
1,2.,4.
5.,NaN,10.0
```

```
In [78]: pd.read_csv(StringIO(data), comment="#", skiprows=4, header=1)
Out[78]:
     A    B     C
0  1.0  2.0   4.0
1  5.0  NaN  10.0
```

Comments

Sometimes comments or meta data may be included in a file:

```
In [79]: data = (
   ....:     "ID,level,category\n"
   ....:     "Patient1,123000,x # really unpleasant\n"
   ....:     "Patient2,23000,y # wouldn't take his medicine\n"
   ....:     "Patient3,1234018,z # awesome"
   ....: )
   ....:

In [80]: with open("tmp.csv", "w") as fh:
   ....:     fh.write(data)
   ....:

In [81]: print(open("tmp.csv").read())
ID,level,category
Patient1,123000,x # really unpleasant
Patient2,23000,y # wouldn't take his medicine
Patient3,1234018,z # awesome
```

By default, the parser includes the comments in the output:

```
In [82]: df = pd.read_csv("tmp.csv")

In [83]: df
Out[83]:
         ID    level                        category
0  Patient1   123000           x # really unpleasant
1  Patient2    23000  y # wouldn't take his medicine
2  Patient3  1234018                     z # awesome
```

We can suppress the comments using the `comment` keyword:

Skip to main content

```
In [85]: df
Out[85]:
         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z
```

Dealing with Unicode data

The `encoding` argument should be used for encoded unicode data, which will result in byte strings being decoded to unicode in the result:

```
In [86]: from io import BytesIO

In [87]: data = b"word,length\n" b"Tr\xc3\xa4umen,7\n" b"Gr\xc3\xbc\xc3\x9fe,5"

In [88]: data = data.decode("utf8").encode("latin-1")

In [89]: df = pd.read_csv(BytesIO(data), encoding="latin-1")

In [90]: df
Out[90]:
      word  length
0  Träumen       7
1    Grüße       5

In [91]: df["word"][1]
Out[91]: 'Grüße'
```

Some formats which encode all characters as multiple bytes, like UTF-16, won't parse correctly at all without specifying the encoding. Full list of Python standard encodings.

Index columns and trailing delimiters

If a file has one more column of data than the number of column names, the first column will be used as the `DataFrame`'s row names:

```
In [92]: data = "a,b,c\n4,apple,bat,5.7\n8,orange,cow,10"

In [93]: pd.read_csv(StringIO(data))
Out[93]:
        a    b     c
4   apple  bat   5.7
8  orange  cow  10.0
```

Skip to main content

```
In [94]: data = "index,a,b,c\n4,apple,bat,5.7\n8,orange,cow,10"

In [95]: pd.read_csv(StringIO(data), index_col=0)
Out[95]:
            a    b     c
index
4       apple  bat   5.7
8      orange  cow  10.0
```

Ordinarily, you can achieve this behavior using the `index_col` option.

There are some exception cases when a file has been prepared with delimiters at the end of each data line, confusing the parser. To explicitly disable the index column inference and discard the last column, pass `index_col=False`:

```
In [96]: data = "a,b,c\n4,apple,bat,\n8,orange,cow,"

In [97]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [98]: pd.read_csv(StringIO(data))
Out[98]:
        a    b   c
4   apple  bat NaN
8  orange  cow NaN

In [99]: pd.read_csv(StringIO(data), index_col=False)
Out[99]:
   a       b    c
0  4   apple  bat
1  8  orange  cow
```

If a subset of data is being parsed using the `usecols` option, the `index_col` specification is based on that subset, not the original data.

```
In [100]: data = "a,b,c\n4,apple,bat,\n8,orange,cow,"

In [101]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [102]: pd.read_csv(StringIO(data), usecols=["b", "c"])
Out[102]:
     b   c
4  bat NaN
8  cow NaN
```

Skip to main content

```
Out[103]:
     b   c
4  bat NaN
8  cow NaN
```

Date Handling

Specifying date columns

To better facilitate working with datetime data, `read_csv()` uses the keyword arguments `parse_dates` and `date_format` to allow users to specify a variety of columns and date/time formats to turn the input text data into `datetime` objects.

The simplest case is to just pass in `parse_dates=True`:

```
In [104]: with open("foo.csv", mode="w") as f:
   .....:     f.write("date,A,B,C\n20090101,a,1,2\n20090102,b,3,4\n20090103,c,4,5")
   .....:

# Use a column as an index, and parse it as dates.
In [105]: df = pd.read_csv("foo.csv", index_col=0, parse_dates=True)

In [106]: df
Out[106]:
            A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5

# These are Python datetime objects
In [107]: df.index
Out[107]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype='datetime64[n
```

It is often the case that we may want to store date and time data separately, or store various date fields separately. the `parse_dates` keyword can be used to specify a combination of columns to parse the dates and/or times from.

You can specify a list of column lists to `parse_dates`, the resulting date columns will be prepended to the output (so as to not affect the existing column order) and the new column names will be the concatenation of the component column names:

```
In [108]: data = (
```

Skip to main content

```
   .....:     "KORD,19990127, 21:00:00, 20:56:00, -0.5900\n"
   .....:     "KORD,19990127, 21:00:00, 21:18:00, -0.9900\n"
   .....:     "KORD,19990127, 22:00:00, 21:56:00, -0.5900\n"
   .....:     "KORD,19990127, 23:00:00, 22:56:00, -0.5900"
   .....: )
   .....:

In [109]: with open("tmp.csv", "w") as fh:
   .....:     fh.write(data)
   .....:

In [110]: df = pd.read_csv("tmp.csv", header=None, parse_dates=[[1, 2], [1, 3]])

In [111]: df
Out[111]:
                  1_2                 1_3     0     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

By default the parser removes the component date columns, but you can choose to retain them via the `keep_date_col` keyword:

```
In [112]: df = pd.read_csv(
   .....:     "tmp.csv", header=None, parse_dates=[[1, 2], [1, 3]], keep_date_col=True
   .....: )
   .....:

In [113]: df
Out[113]:
                  1_2                 1_3     0         1          2          3     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  19990127   19:00:00   18:56:00  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  19990127   20:00:00   19:56:00  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD  19990127   21:00:00   20:56:00 -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD  19990127   21:00:00   21:18:00 -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD  19990127   22:00:00   21:56:00 -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD  19990127   23:00:00   22:56:00 -0.59
```

Note that if you wish to combine multiple columns into a single date column, a nested list must be used. In other words, `parse_dates=[1, 2]` indicates that the second and third columns should each be parsed as separate date columns while `parse_dates=[[1, 2]]` means the two columns should be parsed into a single column.

You can also use a dict to specify custom name columns:

```
In [114]: date_spec = {"nominal": [1, 2], "actual": [1, 3]}
```

Skip to main content

```
In [116]: df
Out[116]:
              nominal              actual     0     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

It is important to remember that if multiple text columns are to be parsed into a single date column, then a new column is prepended to the data. The `index_col` specification is based off of this new set of columns rather than the original data columns:

```
In [117]: date_spec = {"nominal": [1, 2], "actual": [1, 3]}

In [118]: df = pd.read_csv(
   .....:     "tmp.csv", header=None, parse_dates=date_spec, index_col=0
   .....: )  # index is the nominal column
   .....:

In [119]: df
Out[119]:
                                actual     0     4
nominal
1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

Note

If a column or index contains an unparsable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `to_datetime()` after `pd.read_csv`.

Note

read_csv has a fast_path for parsing datetime strings in iso8601 format, e.g “2000-01-01T00:01:02+00:00” and similar variations. If you can arrange for your data to store datetimes in this format, load times will be significantly faster, ~20x has been observed.

Skip to main content

Deprecated since version 2.2.0: Combining date columns inside read_csv is deprecated. Use `pd.to_datetime` on the relevant result columns instead.

Date parsing functions

Finally, the parser allows you to specify a custom `date_format`. Performance-wise, you should try these methods of parsing dates in order:

1. If you know the format, use `date_format`, e.g.: `date_format="%d/%m/%Y"` or `date_format={column_name: "%d/%m/%Y"}`.
2. If you different formats for different columns, or want to pass any extra options (such as `utc`) to `to_datetime`, then you should read in your data as `object` dtype, and then use `to_datetime`.

Parsing a CSV with mixed timezones

pandas cannot natively represent a column or index with mixed timezones. If your CSV file contains columns with a mixture of timezones, the default result will be an object-dtype column with strings, even with `parse_dates`. To parse the mixed-timezone values as a datetime column, read in as `object` dtype and then call `to_datetime()` with `utc=True`.

```
In [120]: content = """\
   .....: a
   .....: 2000-01-01T00:00:00+05:00
   .....: 2000-01-01T00:00:00+06:00"""
   .....:

In [121]: df = pd.read_csv(StringIO(content))

In [122]: df["a"] = pd.to_datetime(df["a"], utc=True)

In [123]: df["a"]
Out[123]:
0   1999-12-31 19:00:00+00:00
1   1999-12-31 18:00:00+00:00
Name: a, dtype: datetime64[ns, UTC]
```

Inferring datetime format

Here are some examples of datetime strings that can be guessed (all representing December 30th, 2011 at 00:00:00):

Skip to main content

- “20111230”
- “2011/12/30”
- “20111230 00:00:00”
- “12/30/2011 00:00:00”
- “30/Dec/2011 00:00:00”
- “30/December/2011 00:00:00”

Note that format inference is sensitive to `dayfirst`. With `dayfirst=True`, it will guess “01/12/2011” to be December 1st. With `dayfirst=False` (default) it will guess “01/12/2011” to be January 12th.

If you try to parse a column of date strings, pandas will attempt to guess the format from the first non-NaN element, and will then parse the rest of the column with that format. If pandas fails to guess the format (for example if your first string is `'01 December US/Pacific 2000'`), then a warning will be raised and each row will be parsed individually by `dateutil.parser.parse`. The safest way to parse dates is to explicitly set `format=`.

```
In [124]: df = pd.read_csv(
   .....:     "foo.csv",
   .....:     index_col=0,
   .....:     parse_dates=True,
   .....: )
   .....:

In [125]: df
Out[125]:
            A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5
```

In the case that you have mixed datetime formats within the same column, you can pass `format='mixed'`

```
In [126]: data = StringIO("date\n12 Jan 2000\n2000-01-13\n")

In [127]: df = pd.read_csv(data)

In [128]: df['date'] = pd.to_datetime(df['date'], format='mixed')

In [129]: df
Out[129]:
        date
0 2000-01-12
```

Skip to main content

or, if your datetime formats are all ISO8601 (possibly not identically-formatted):

```
In [130]: data = StringIO("date\n2020-01-01\n2020-01-01 03:00\n")

In [131]: df = pd.read_csv(data)

In [132]: df['date'] = pd.to_datetime(df['date'], format='ISO8601')

In [133]: df
Out[133]:
                 date
0 2020-01-01 00:00:00
1 2020-01-01 03:00:00
```

International date formats

While US date formats tend to be MM/DD/YYYY, many international formats use DD/MM/YYYY instead. For convenience, a `dayfirst` keyword is provided:

```
In [134]: data = "date,value,cat\n1/6/2000,5,a\n2/6/2000,10,b\n3/6/2000,15,c"

In [135]: print(data)
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c

In [136]: with open("tmp.csv", "w") as fh:
   .....:     fh.write(data)
   .....:

In [137]: pd.read_csv("tmp.csv", parse_dates=[0])
Out[137]:
        date  value cat
0 2000-01-06      5   a
1 2000-02-06     10   b
2 2000-03-06     15   c

In [138]: pd.read_csv("tmp.csv", dayfirst=True, parse_dates=[0])
Out[138]:
        date  value cat
0 2000-06-01      5   a
1 2000-06-02     10   b
2 2000-06-03     15   c
```

Writing CSVs to binary file objects

Skip to main content

`df.to_csv(..., mode="wb")` allows writing a CSV to a file object opened binary mode. In most cases, it is not necessary to specify `mode` as Pandas will auto-detect whether the file object is opened in text or binary mode.

```
In [139]: import io

In [140]: data = pd.DataFrame([0, 1, 2])

In [141]: buffer = io.BytesIO()

In [142]: data.to_csv(buffer, encoding="utf-8", compression="gzip")
```

Specifying method for floating-point conversion

The parameter `float_precision` can be specified in order to use a specific floating-point converter during parsing with the C engine. The options are the ordinary converter, the high-precision converter, and the round-trip converter (which is guaranteed to round-trip values after writing to a file). For example:

```
In [143]: val = "0.3066101993807095471566981359501369297504425048828125"

In [144]: data = "a,b,c\n1,2,{0}".format(val)

In [145]: abs(
   .....:     pd.read_csv(
   .....:         StringIO(data),
   .....:         engine="c",
   .....:         float_precision=None,
   .....:     )["c"][0] - float(val)
   .....: )
   .....:
Out[145]: 5.551115123125783e-17

In [146]: abs(
   .....:     pd.read_csv(
   .....:         StringIO(data),
   .....:         engine="c",
   .....:         float_precision="high",
   .....:     )["c"][0] - float(val)
   .....: )
   .....:
Out[146]: 5.551115123125783e-17

In [147]: abs(
   .....:     pd.read_csv(StringIO(data), engine="c", float_precision="round_trip")["c"
   .....:     - float(val)
   .....: )
   .....:
Out[147]: 0.0
```

Skip to main content

Thousand separators

For large numbers that have been written with a thousands separator, you can set the `thousands` keyword to a string of length 1 so that integers will be parsed correctly:

By default, numbers with a thousands separator will be parsed as strings:

```
In [148]: data = (
   .....:     "ID|level|category\n"
   .....:     "Patient1|123,000|x\n"
   .....:     "Patient2|23,000|y\n"
   .....:     "Patient3|1,234,018|z"
   .....: )
   .....:

In [149]: with open("tmp.csv", "w") as fh:
   .....:     fh.write(data)
   .....:

In [150]: df = pd.read_csv("tmp.csv", sep="|")

In [151]: df
Out[151]:
         ID      level category
0  Patient1    123,000        x
1  Patient2     23,000        y
2  Patient3  1,234,018        z

In [152]: df.level.dtype
Out[152]: dtype('O')
```

The `thousands` keyword allows integers to be parsed correctly:

```
In [153]: df = pd.read_csv("tmp.csv", sep="|", thousands=",")

In [154]: df
Out[154]:
         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z

In [155]: df.level.dtype
Out[155]: dtype('int64')
```

NA values

To control which values are parsed as missing values (which are signified by `NaN`), specify a string

Skip to main content

values. If you specify a number (a `float`, like `5.0` or an `integer` like `5`), the corresponding equivalent values will also imply a missing value (in this case effectively `[5.0, 5]` are recognized as `NaN`).

To completely override the default values that are recognized as missing, specify `keep_default_na=False`.

The default `NaN` recognized values are `['-1.#IND', '1.#QNAN', '1.#IND', '-1.#QNAN', '#N/A N/A', '#N/A', 'N/A', 'n/a', 'NA', '<NA>', '#NA', 'NULL', 'null', 'NaN', '-NaN', 'nan', '-nan', 'None', '']`.

Let us consider some examples:

```
pd.read_csv("path_to_file.csv", na_values=[5])
```

In the example above `5` and `5.0` will be recognized as `NaN`, in addition to the defaults. A string will first be interpreted as a numerical `5`, then as a `NaN`.

```
pd.read_csv("path_to_file.csv", keep_default_na=False, na_values=[""])
```

Above, only an empty field will be recognized as `NaN`.

```
pd.read_csv("path_to_file.csv", keep_default_na=False, na_values=["NA", "0"])
```

Above, both `NA` and `0` as strings are `NaN`.

```
pd.read_csv("path_to_file.csv", na_values=["Nope"])
```

The default values, in addition to the string `"Nope"` are recognized as `NaN`.

Infinity

`inf` like values will be parsed as `np.inf` (positive infinity), and `-inf` as `-np.inf` (negative infinity). These will ignore the case of the value, meaning `Inf`, will also be parsed as `np.inf`.

Boolean values

Skip to main content

The common values `True`, `False`, `TRUE`, and `FALSE` are all recognized as boolean. Occasionally you might want to recognize other values as being boolean. To do this, use the `true_values` and `false_values` options as follows:

```
In [156]: data = "a,b,c\n1,Yes,2\n3,No,4"

In [157]: print(data)
a,b,c
1,Yes,2
3,No,4

In [158]: pd.read_csv(StringIO(data))
Out[158]:
   a    b  c
0  1  Yes  2
1  3   No  4

In [159]: pd.read_csv(StringIO(data), true_values=["Yes"], false_values=["No"])
Out[159]:
   a      b  c
0  1   True  2
1  3  False  4
```

Handling “bad” lines

Some files may have malformed lines with too few fields or too many. Lines with too few fields will have NA values filled in the trailing fields. Lines with too many fields will raise an error by default:

```
In [160]: data = "a,b,c\n1,2,3\n4,5,6,7\n8,9,10"

In [161]: pd.read_csv(StringIO(data))
---------------------------------------------------------------------------
ParserError                               Traceback (most recent call last)
Cell In[161], line 1
----> 1 pd.read_csv(StringIO(data))

File /home/pandas/pandas/io/parsers/readers.py:1024, in read_csv(filepath_or_buffer, se
   1011 kwds_defaults = _refine_defaults_read(
   1012     dialect,
   1013     delimiter,
   (...)
   1020     dtype_backend=dtype_backend,
   1021 )
   1022 kwds.update(kwds_defaults)
-> 1024 return _read(filepath_or_buffer, kwds)

File /home/pandas/pandas/io/parsers/readers.py:624, in _read(filepath_or_buffer, kwds)
    621     return parser
```

Skip to main content

```
File /home/pandas/pandas/io/parsers/readers.py:1921, in TextFileReader.read(self, nrows
   1914 nrows = validate_integer("nrows", nrows)
   1915 try:
   1916     # error: "ParserBase" has no attribute "read"
   1917     (
   1918         index,
   1919         columns,
   1920         col_dict,
-> 1921     ) = self._engine.read(  # type: ignore[attr-defined]
   1922         nrows
   1923     )
   1924 except Exception:
   1925     self.close()

File /home/pandas/pandas/io/parsers/c_parser_wrapper.py:234, in CParserWrapper.read(sel
    232 try:
    233     if self.low_memory:
--> 234         chunks = self._reader.read_low_memory(nrows)
    235         # destructive to chunks
    236         data = _concatenate_chunks(chunks)

File /home/pandas/pandas/_libs/parsers.pyx:838, in pandas._libs.parsers.TextReader.read

File /home/pandas/pandas/_libs/parsers.pyx:905, in pandas._libs.parsers.TextReader._rea

File /home/pandas/pandas/_libs/parsers.pyx:874, in pandas._libs.parsers.TextReader._tok

File /home/pandas/pandas/_libs/parsers.pyx:891, in pandas._libs.parsers.TextReader._che

File /home/pandas/pandas/_libs/parsers.pyx:2061, in pandas._libs.parsers.raise_parser_e

ParserError: Error tokenizing data. C error: Expected 3 fields in line 3, saw 4
```

You can elect to skip bad lines:

```
In [162]: data = "a,b,c\n1,2,3\n4,5,6,7\n8,9,10"

In [163]: pd.read_csv(StringIO(data), on_bad_lines="skip")
Out[163]:
   a  b   c
0  1  2   3
1  8  9  10
```

New in version 1.4.0.

Or pass a callable function to handle the bad line if `engine="python"`. The bad line will be a list of strings that was split by the `sep`:

Skip to main content

```
In [165]: def bad_lines_func(line):
   .....:     external_list.append(line)
   .....:     return line[-3:]
   .....:

In [166]: external_list
Out[166]: []
```

Note

The callable function will handle only a line with too many fields. Bad lines caused by other errors will be silently skipped.

```
In [167]: bad_lines_func = lambda line: print(line)

In [168]: data = 'name,type\nname a,a is of type a\nname b,"b\" is of type b"'

In [169]: data
Out[169]: 'name,type\nname a,a is of type a\nname b,"b" is of type b"'

In [170]: pd.read_csv(StringIO(data), on_bad_lines=bad_lines_func, engine="pyt
Out[170]:
     name            type
0  name a  a is of type a
```

The line was not processed in this case, as a “bad line” here is caused by an escape character.

You can also use the `usecols` parameter to eliminate extraneous column data that appear in some lines but not others:

```
In [171]: pd.read_csv(StringIO(data), usecols=[0, 1, 2])
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[171], line 1
----> 1 pd.read_csv(StringIO(data), usecols=[0, 1, 2])

File /home/pandas/pandas/io/parsers/readers.py:1024, in read_csv(filepath_or_buffer, se
   1011 kwds_defaults = _refine_defaults_read(
   1012     dialect,
   1013     delimiter,
   (...)
   1020     dtype_backend=dtype_backend,
   1021 )
   1022 kwds.update(kwds_defaults)
-> 1024 return _read(filepath_or_buffer, kwds)
```

Skip to main content

```
    617 # Create the parser.
--> 618 parser = TextFileReader(filepath_or_buffer, **kwds)
    620 if chunksize or iterator:
    621     return parser

File /home/pandas/pandas/io/parsers/readers.py:1618, in TextFileReader.__init__(self, f
   1615     self.options["has_index_names"] = kwds["has_index_names"]
   1617 self.handles: IOHandles | None = None
-> 1618 self._engine = self._make_engine(f, self.engine)

File /home/pandas/pandas/io/parsers/readers.py:1896, in TextFileReader._make_engine(sel
   1893     raise ValueError(msg)
   1895 try:
-> 1896     return mapping[engine](f, **self.options)
   1897 except Exception:
   1898     if self.handles is not None:

File /home/pandas/pandas/io/parsers/c_parser_wrapper.py:155, in CParserWrapper.__init__
    152     # error: Cannot determine type of 'names'
    153     if len(self.names) < len(usecols):  # type: ignore[has-type]
    154         # error: Cannot determine type of 'names'
--> 155         self._validate_usecols_names(
    156             usecols,
    157             self.names,  # type: ignore[has-type]
    158         )
    160 # error: Cannot determine type of 'names'
    161 self._validate_parse_dates_presence(self.names)  # type: ignore[has-type]

File /home/pandas/pandas/io/parsers/base_parser.py:979, in ParserBase._validate_usecols
    977 missing = [c for c in usecols if c not in names]
    978 if len(missing) > 0:
--> 979     raise ValueError(
    980         f"Usecols do not match columns, columns expected but not found: "
    981         f"{missing}"
    982     )
    984 return usecols

ValueError: Usecols do not match columns, columns expected but not found: [0, 1, 2]
```

In case you want to keep all data including the lines with too many fields, you can specify a sufficient number of `names`. This ensures that lines with not enough fields are filled with `NaN`.

```
In [172]: pd.read_csv(StringIO(data), names=['a', 'b', 'c', 'd'])
Out[172]:
        a               b   c   d
0    name            type NaN NaN
1  name a   a is of type a NaN NaN
2  name b  b is of type b" NaN NaN
```

Dialect

Skip to main content

The `dialect` keyword gives greater flexibility in specifying the file format. By default it uses the Excel dialect but you can specify either the dialect name or a `csv.Dialect` instance.

Suppose you had data with unenclosed quotes:

```
In [173]: data = "label1,label2,label3\n" 'index1,"a,c,e\n' "index2,b,d,f"

In [174]: print(data)
label1,label2,label3
index1,"a,c,e
index2,b,d,f
```

By default, `read_csv` uses the Excel dialect and treats the double quote as the quote character, which causes it to fail when it finds a newline before it finds the closing double quote.

We can get around this using `dialect`:

```
In [175]: import csv

In [176]: dia = csv.excel()

In [177]: dia.quoting = csv.QUOTE_NONE

In [178]: pd.read_csv(StringIO(data), dialect=dia)
Out[178]:
       label1 label2 label3
index1     "a      c      e
index2      b      d      f
```

All of the dialect options can be specified separately by keyword arguments:

```
In [179]: data = "a,b,c~1,2,3~4,5,6"

In [180]: pd.read_csv(StringIO(data), lineterminator="~")
Out[180]:
   a  b  c
0  1  2  3
1  4  5  6
```

Another common dialect option is `skipinitialspace`, to skip any whitespace after a delimiter:

```
In [181]: data = "a, b, c\n1, 2, 3\n4, 5, 6"

In [182]: print(data)
a, b, c
1, 2, 3
4, 5, 6
```

Skip to main content

```
In [183]: pd.read_csv(StringIO(data), skipinitialspace=True)
Out[183]:
   a  b  c
0  1  2  3
1  4  5  6
```

The parsers make every attempt to "do the right thing" and not be fragile. Type inference is a pretty big deal. If a column can be coerced to integer dtype without altering the contents, the parser will do so. Any non-numeric columns will come through as object dtype as with the rest of pandas objects.

Quoting and Escape Characters

Quotes (and other escape characters) in embedded fields can be handled in any number of ways. One way is to use backslashes; to properly parse this data, you should pass the `escapechar` option:

```
In [184]: data = 'a,b\n"hello, \\"Bob\\", nice to see you",5'

In [185]: print(data)
a,b
"hello, \"Bob\", nice to see you",5

In [186]: pd.read_csv(StringIO(data), escapechar="\\")
Out[186]:
                              a  b
0  hello, "Bob", nice to see you  5
```

Files with fixed width columns

While `read_csv()` reads delimited data, the `read_fwf()` function works with data files that have known and fixed column widths. The function parameters to `read_fwf` are largely the same as `read_csv` with two extra parameters, and a different usage of the `delimiter` parameter:

- `colspecs`: A list of pairs (tuples) giving the extents of the fixed-width fields of each line as half-open intervals (i.e., [from, to[ ). String value 'infer' can be used to instruct the parser to try detecting the column specifications from the first 100 rows of the data. Default behavior, if not specified, is to infer.
- `widths`: A list of field widths which can be used instead of 'colspecs' if the intervals are contiguous.
- `delimiter`: Characters to consider as filler characters in the fixed-width file. Can be used to

Skip to main content

Consider a typical fixed-width data file:

```
In [187]: data1 = (
   .....:     "id8141    360.242940   149.910199   11950.7\n"
   .....:     "id1594    444.953632   166.985655   11788.4\n"
   .....:     "id1849    364.136849   183.628767   11806.2\n"
   .....:     "id1230    413.836124   184.375703   11916.8\n"
   .....:     "id1948    502.953953   173.237159   12468.3"
   .....: )
   .....:

In [188]: with open("bar.csv", "w") as f:
   .....:     f.write(data1)
   .....:
```

In order to parse this file into a `DataFrame`, we simply need to supply the column specifications to the `read_fwf` function along with the file name:

```
# Column specifications are a list of half-intervals
In [189]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [190]: df = pd.read_fwf("bar.csv", colspecs=colspecs, header=None, index_col=0)

In [191]: df
Out[191]:
                1           2        3
0
id8141  360.242940  149.910199  11950.7
id1594  444.953632  166.985655  11788.4
id1849  364.136849  183.628767  11806.2
id1230  413.836124  184.375703  11916.8
id1948  502.953953  173.237159  12468.3
```

Note how the parser automatically picks column names X.<column number> when `header=None` argument is specified. Alternatively, you can supply just the column widths for contiguous columns:

```
# Widths are a list of integers
In [192]: widths = [6, 14, 13, 10]

In [193]: df = pd.read_fwf("bar.csv", widths=widths, header=None)

In [194]: df
Out[194]:
        0           1           2        3
0  id8141  360.242940  149.910199  11950.7
1  id1594  444.953632  166.985655  11788.4
2  id1849  364.136849  183.628767  11806.2
3  id1230  413.836124  184.375703  11916.8
4  id1948  502.953953  173.237159  12468.3
```

Skip to main content

The parser will take care of extra white spaces around the columns so it's ok to have extra separation between the columns in the file.

By default, `read_fwf` will try to infer the file's `colspecs` by using the first 100 rows of the file. It can do it only in cases when the columns are aligned and correctly separated by the provided `delimiter` (default delimiter is whitespace).

```
In [195]: df = pd.read_fwf("bar.csv", header=None, index_col=0)

In [196]: df
Out[196]:
                 1           2        3
0
id8141  360.242940  149.910199  11950.7
id1594  444.953632  166.985655  11788.4
id1849  364.136849  183.628767  11806.2
id1230  413.836124  184.375703  11916.8
id1948  502.953953  173.237159  12468.3
```

`read_fwf` supports the `dtype` parameter for specifying the types of parsed columns to be different from the inferred type.

```
In [197]: pd.read_fwf("bar.csv", header=None, index_col=0).dtypes
Out[197]:
1    float64
2    float64
3    float64
dtype: object

In [198]: pd.read_fwf("bar.csv", header=None, dtype={2: "object"}).dtypes
Out[198]:
0     object
1    float64
2     object
3    float64
dtype: object
```

Indexes

Files with an "implicit" index column

Consider a file with one less entry in the header than the number of data column:

```
In [199]: data = "A,B,C\n20090101,a,1,2\n20090102,b,3,4\n20090103,c,4,5"
```

Skip to main content

```
A,B,C
20090101,a,1,2
20090102,b,3,4
20090103,c,4,5

In [201]: with open("foo.csv", "w") as f:
   .....:     f.write(data)
   .....:
```

In this special case, `read_csv` assumes that the first column is to be used as the index of the `DataFrame`:

```
In [202]: pd.read_csv("foo.csv")
Out[202]:
          A  B  C
20090101  a  1  2
20090102  b  3  4
20090103  c  4  5
```

Note that the dates weren't automatically parsed. In that case you would need to do as before:

```
In [203]: df = pd.read_csv("foo.csv", parse_dates=True)

In [204]: df.index
Out[204]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype='datetime64[n
```

Reading an index with a `MultiIndex`

Suppose you have data indexed by two columns:

```
In [205]: data = 'year,indiv,zit,xit\n1977,"A",1.2,.6\n1977,"B",1.5,.5'

In [206]: print(data)
year,indiv,zit,xit
1977,"A",1.2,.6
1977,"B",1.5,.5

In [207]: with open("mindex_ex.csv", mode="w") as f:
   .....:     f.write(data)
   .....:
```

The `index_col` argument to `read_csv` can take a list of column numbers to turn multiple columns into a `MultiIndex` for the index of the returned object:

Skip to main content

```
In [209]: df
Out[209]:
            zit  xit
year indiv
1977 A      1.2  0.6
     B      1.5  0.5

In [210]: df.loc[1977]
Out[210]:
       zit  xit
indiv
A      1.2  0.6
B      1.5  0.5
```

Reading columns with a `MultiIndex`

By specifying list of row locations for the `header` argument, you can read in a `MultiIndex` for the columns. Specifying non-consecutive rows will skip the intervening rows.

```
In [211]: mi_idx = pd.MultiIndex.from_arrays([[1, 2, 3, 4], list("abcd")], names=list("

In [212]: mi_col = pd.MultiIndex.from_arrays([[1, 2], list("ab")], names=list("cd"))

In [213]: df = pd.DataFrame(np.ones((4, 2)), index=mi_idx, columns=mi_col)

In [214]: df.to_csv("mi.csv")

In [215]: print(open("mi.csv").read())
c,,1,2
d,,a,b
a,b,,
1,a,1.0,1.0
2,b,1.0,1.0
3,c,1.0,1.0
4,d,1.0,1.0


In [216]: pd.read_csv("mi.csv", header=[0, 1, 2, 3], index_col=[0, 1])
Out[216]:
c                    1                   2
d                    a                   b
a   Unnamed: 2_level_2 Unnamed: 3_level_2
1                  1.0                 1.0
2 b                1.0                 1.0
3 c                1.0                 1.0
4 d                1.0                 1.0
```

`read_csv` is also able to interpret a more common format of multi-columns indices.

Skip to main content

```
In [218]: print(data)
,a,a,a,b,c,c
,q,r,s,t,u,v
one,1,2,3,4,5,6
two,7,8,9,10,11,12

In [219]: with open("mi2.csv", "w") as fh:
   .....:     fh.write(data)
   .....:

In [220]: pd.read_csv("mi2.csv", header=[0, 1], index_col=0)
Out[220]:
     a         b   c
     q  r  s   t   u   v
one  1  2  3   4   5   6
two  7  8  9  10  11  12
```

Note

If an `index_col` is not specified (e.g. you don't have an index, or wrote it with `df.to_csv(..., index=False)`, then any `names` on the columns index will be *lost*.

Automatically "sniffing" the delimiter

`read_csv` is capable of inferring delimited (not necessarily comma-separated) files, as pandas uses the `csv.Sniffer` class of the csv module. For this, you have to specify `sep=None`.

```
In [221]: df = pd.DataFrame(np.random.randn(10, 4))

In [222]: df.to_csv("tmp2.csv", sep=":", index=False)

In [223]: pd.read_csv("tmp2.csv", sep=None, engine="python")
Out[223]:
          0         1         2         3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209 -1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
7 -0.370647 -1.157892 -1.344312  0.844885
8  1.075770 -0.109050  1.643563 -1.469388
9  0.357021 -0.674600 -1.776904 -0.968914
```

Reading multiple files to create a single DataFrame

Skip to main content

It's best to use `concat()` to combine multiple files. See the cookbook for an example.

Iterating through files chunk by chunk

Suppose you wish to iterate through a (potentially very large) file lazily rather than reading the entire file into memory, such as the following:

```
In [224]: df = pd.DataFrame(np.random.randn(10, 4))

In [225]: df.to_csv("tmp.csv", index=False)

In [226]: table = pd.read_csv("tmp.csv")

In [227]: table
Out[227]:
          0         1         2         3
0 -1.294524  0.413738  0.276662 -0.472035
1 -0.013960 -0.362543 -0.006154 -0.923061
2  0.895717  0.805244 -1.206412  2.565646
3  1.431256  1.340309 -1.170299 -0.226169
4  0.410835  0.813850  0.132003 -0.827317
5 -0.076467 -1.187678  1.130127 -1.436737
6 -1.413681  1.607920  1.024180  0.569605
7  0.875906 -2.211372  0.974466 -2.006747
8 -0.410001 -0.078638  0.545952 -1.219217
9 -1.226825  0.769804 -1.281247 -0.727707
```

By specifying a `chunksize` to `read_csv`, the return value will be an iterable object of type `TextFileReader`:

```
In [228]: with pd.read_csv("tmp.csv", chunksize=4) as reader:
   .....:     print(reader)
   .....:     for chunk in reader:
   .....:         print(chunk)
   .....:
<pandas.io.parsers.readers.TextFileReader object at 0x7efe79bdc850>
          0         1         2         3
0 -1.294524  0.413738  0.276662 -0.472035
1 -0.013960 -0.362543 -0.006154 -0.923061
2  0.895717  0.805244 -1.206412  2.565646
3  1.431256  1.340309 -1.170299 -0.226169
          0         1         2         3
4  0.410835  0.813850  0.132003 -0.827317
5 -0.076467 -1.187678  1.130127 -1.436737
6 -1.413681  1.607920  1.024180  0.569605
7  0.875906 -2.211372  0.974466 -2.006747
          0         1         2         3
8 -0.410001 -0.078638  0.545952 -1.219217
9 -1.226825  0.769804 -1.281247 -0.727707
```

Skip to main content

Changed in version 1.2: `read_csv/json/sas` return a context-manager when iterating through a file.

Specifying `iterator=True` will also return the `TextFileReader` object:

```
In [229]: with pd.read_csv("tmp.csv", iterator=True) as reader:
   .....:     print(reader.get_chunk(5))
   .....:
          0         1         2         3
0 -1.294524  0.413738  0.276662 -0.472035
1 -0.013960 -0.362543 -0.006154 -0.923061
2  0.895717  0.805244 -1.206412  2.565646
3  1.431256  1.340309 -1.170299 -0.226169
4  0.410835  0.813850  0.132003 -0.827317
```

Specifying the parser engine

Pandas currently supports three engines, the C engine, the python engine, and an experimental pyarrow engine (requires the `pyarrow` package). In general, the pyarrow engine is fastest on larger workloads and is equivalent in speed to the C engine on most other workloads. The python engine tends to be slower than the pyarrow and C engines on most workloads. However, the pyarrow engine is much less robust than the C engine, which lacks a few features compared to the Python engine.

Where possible, pandas uses the C parser (specified as `engine='c'`), but it may fall back to Python if C-unsupported options are specified.

Currently, options unsupported by the C and pyarrow engines include:

- `sep` other than a single character (e.g. regex separators)
- `skipfooter`
- `sep=None` with `delim_whitespace=False`

Specifying any of the above options will produce a `ParserWarning` unless the python engine is selected explicitly using `engine='python'`.

Options that are unsupported by the pyarrow engine which are not covered by the list above include:

- `float_precision`

Skip to main content

- `comment`
- `nrows`
- `thousands`
- `memory_map`
- `dialect`
- `on_bad_lines`
- `delim_whitespace`
- `quoting`
- `lineterminator`
- `converters`
- `decimal`
- `iterator`
- `dayfirst`
- `infer_datetime_format`
- `verbose`
- `skipinitialspace`
- `low_memory`

Specifying these options with `engine='pyarrow'` will raise a `ValueError`.

Reading/writing remote files

You can pass in a URL to read or write remote files to many of pandas' IO functions - the following example shows reading a CSV file:

```
df = pd.read_csv("https://download.bls.gov/pub/time.series/cu/cu.item", sep="\t")
```

> ***New in version 1.3.0.***

A custom header can be sent alongside HTTP(s) requests by passing a dictionary of header key value mappings to the `storage_options` keyword argument as shown below:

```
headers = {"User-Agent": "pandas"}
df = pd.read_csv(
    "https://download.bls.gov/pub/time.series/cu/cu.item",
```

Skip to main content

```
    storage_options=headers
)
```

All URLs which are not local files or HTTP(s) are handled by fsspec, if installed, and its various filesystem implementations (including Amazon S3, Google Cloud, SSH, FTP, webHDFS...). Some of these implementations will require additional packages to be installed, for example S3 URLs require the s3fs library:

```
df = pd.read_json("s3://pandas-test/adatafile.json")
```

When dealing with remote storage systems, you might need extra configuration with environment variables or config files in special locations. For example, to access data in your S3 bucket, you will need to define credentials in one of the several ways listed in the S3Fs documentation. The same is true for several of the storage backends, and you should follow the links at fsimpl1 for implementations built into `fsspec` and fsimpl2 for those not included in the main `fsspec` distribution.

You can also pass parameters directly to the backend driver. Since `fsspec` does not utilize the `AWS_S3_HOST` environment variable, we can directly define a dictionary containing the endpoint_url and pass the object into the storage option parameter:

```
storage_options = {"client_kwargs": {"endpoint_url": "http://127.0.0.1:5555"}}}
df = pd.read_json("s3://pandas-test/test-1", storage_options=storage_options)
```

More sample configurations and documentation can be found at S3Fs documentation.

If you do *not* have S3 credentials, you can still access public data by specifying an anonymous connection, such as

> **New in version 1.2.0.**

```
>>> df = pd.read_csv(
...     "s3://ncei-wcsd-archive/data/processed/SH1305/18kHz/SaKe2013"
...     "-D20130523-T080854_to_SaKe2013-D20130523-T085643.csv",
...     storage_options={"anon": True},
...)
>>> df.columns
Index(['Ping_index', ' Distance_gps', ' Distance_vl', ' Ping_date',
       ' Ping_time', ' Ping_milliseconds', ' Latitude', ' Longitude',
       ' Depth_start', ' Depth_stop', ' Range_start', ' Range_stop',
       ' Sample_count'],
```

Skip to main content

`fsspec` also allows complex URLs, for accessing data in compressed archives, local caching of files, and more. To locally cache the above example, you would modify the call to

```
pd.read_csv(
    "simplecache::s3://ncei-wcsd-archive/data/processed/SH1305/18kHz/"
    "SaKe2013-D20130523-T080854_to_SaKe2013-D20130523-T085643.csv",
    storage_options={"s3": {"anon": True}},
)
```

where we specify that the "anon" parameter is meant for the "s3" part of the implementation, not to the caching implementation. Note that this caches to a temporary directory for the duration of the session only, but you can also specify a permanent store.

Writing out data

Writing to CSV format

The `Series` and `DataFrame` objects have an instance method `to_csv` which allows storing the contents of the object as a comma-separated-values file. The function takes a number of arguments. Only the first is required.

- `path_or_buf`: A string path to the file to write or a file object. If a file object it must be opened with `newline=''`
- `sep` : Field delimiter for the output file (default ",")
- `na_rep`: A string representation of a missing value (default '')
- `float_format`: Format string for floating point numbers
- `columns`: Columns to write (default None)
- `header`: Whether to write out the column names (default True)
- `index`: whether to write row (index) names (default True)
- `index_label`: Column label(s) for index column(s) if desired. If None (default), and `header` and `index` are True, then the index names are used. (A sequence should be given if the `DataFrame` uses MultiIndex).
- `mode` : Python write mode, default 'w'
- `encoding`: a string representing the encoding to use if the contents are non-ASCII, for Python versions prior to 3
- `lineterminator`: Character sequence denoting line end (default `os.linesep`)
- `quoting`: Set quoting rules as in csv module (default csv.QUOTE_MINIMAL). Note that if you

Skip to main content

will treat them as non-numeric

- `quotechar`: Character used to quote fields (default '"')
- `doublequote`: Control quoting of `quotechar` in fields (default True)
- `escapechar`: Character used to escape `sep` and `quotechar` when appropriate (default None)
- `chunksize`: Number of rows to write at a time
- `date_format`: Format string for datetime objects

Writing a formatted string

The `DataFrame` object has an instance method `to_string` which allows control over the string representation of the object. All arguments are optional:

- `buf` default None, for example a StringIO object
- `columns` default None, which columns to write
- `col_space` default None, minimum width of each column.
- `na_rep` default `NaN`, representation of NA value
- `formatters` default None, a dictionary (by column) of functions each of which takes a single argument and returns a formatted string
- `float_format` default None, a function which takes a single (float) argument and returns a formatted string; to be applied to floats in the `DataFrame`.
- `sparsify` default True, set to False for a `DataFrame` with a hierarchical index to print every MultiIndex key at each row.
- `index_names` default True, will print the names of the indices
- `index` default True, will print the index (ie, row labels)
- `header` default True, will print the column labels
- `justify` default `left`, will print column headers left- or right-justified

The `Series` object also has a `to_string` method, but with only the `buf`, `na_rep`, `float_format` arguments. There is also a `length` argument which, if set to `True`, will additionally output the length of the Series.

JSON

Read and write `JSON` format files and strings.

Skip to main content

Writing JSON

A `Series` or `DataFrame` can be converted to a valid JSON string. Use `to_json` with optional parameters:

- `path_or_buf` : the pathname or buffer to write the output. This can be `None` in which case a JSON string is returned.
- `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

- `date_format` : string, type of date conversion, ‘epoch’ for timestamp, ‘iso’ for ISO8601.
- `double_precision` : The number of decimal places to use when encoding floating point values, default 10.
- `force_ascii` : force encoded string to be ASCII, default True.
- `date_unit` : The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’ or ‘ns’ for seconds, milliseconds, microseconds and nanoseconds respectively. Default ‘ms’.
- `default_handler` : The handler to call if an object cannot otherwise be converted to a suitable format for JSON. Takes a single argument, which is the object to convert, and returns a serializable object.

Skip to main content

- `mode` : string, writer mode when writing to path. 'w' for write, 'a' for append. Default 'w'

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 [230]: dfj = pd.DataFrame(np.random.randn(5, 2), columns=list("AB"))

In [231]: json = dfj.to_json()

In [232]: json
Out[232]: '{"A":{"0":-0.1213062281,"1":0.6957746499,"2":0.9597255933,"3":-0.6199759194,
```

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

In [234]: dfjo
Out[234]:
   A  B  C
x  1  4  7
y  2  5  8
z  3  6  9

In [235]: sjo = pd.Series(dict(x=15, y=16, z=17), name="D")

In [236]: sjo
Out[236]:
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 [237]: dfjo.to_json(orient="columns")
Out[237]: '{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":{"x":7,"y":8,"z":9}}'
```

Skip to main content

```
# Not available for Series
```

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

```
In [238]: dfjo.to_json(orient="index")
Out[238]: '{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":{"A":3,"B":6,"C":9}}'

In [239]: sjo.to_json(orient="index")
Out[239]: '{"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 [240]: dfjo.to_json(orient="records")
Out[240]: '[{"A":1,"B":4,"C":7},{"A":2,"B":5,"C":8},{"A":3,"B":6,"C":9}]'

In [241]: sjo.to_json(orient="records")
Out[241]: '[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 [242]: dfjo.to_json(orient="values")
Out[242]: '[[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 [243]: dfjo.to_json(orient="split")
Out[243]: '{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,5,8],[3,6,

In [244]: sjo.to_json(orient="split")
Out[244]: '{"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.

Skip to main content

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 [245]: dfd = pd.DataFrame(np.random.randn(5, 2), columns=list("AB"))

In [246]: dfd["date"] = pd.Timestamp("20130101")

In [247]: dfd = dfd.sort_index(axis=1, ascending=False)

In [248]: json = dfd.to_json(date_format="iso")

In [249]: json
Out[249]: '{"date":{"0":"2013-01-01T00:00:00.000","1":"2013-01-01T00:00:00.000","2":"20
```

Writing in ISO date format, with microseconds:

```
In [250]: json = dfd.to_json(date_format="iso", date_unit="us")

In [251]: json
Out[251]: '{"date":{"0":"2013-01-01T00:00:00.000000","1":"2013-01-01T00:00:00.000000","
```

Epoch timestamps, in seconds:

```
In [252]: json = dfd.to_json(date_format="epoch", date_unit="s")

In [253]: json
Out[253]: '{"date":{"0":1,"1":1,"2":1,"3":1,"4":1},"B":{"0":0.403309524,"1":0.301624452
```

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

```
In [254]: dfj2 = dfj.copy()

In [255]: dfj2["date"] = pd.Timestamp("20130101")
```

Skip to main content

```
In [257]: dfj2["bools"] = True

In [258]: dfj2.index = pd.date_range("20130101", periods=5)

In [259]: dfj2.to_json("test.json")

In [260]: with open("test.json") as fh:
   .....:     print(fh.read())
   .....:
{"A":{"1356998400000":-0.1213062281,"1357084800000":0.6957746499,"1357171200000":0.9597
```

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 [261]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
Out[261]: '{"0":{"0":"(1+0j)","1":"(2+0j)","2":"(1+2j)"}}'
```

Reading JSON

Skip to main content

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.
- `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

Skip to main content

- `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 `pandas.api.typing.JsonReader` which reads in `chunksize` lines per iteration.
- `engine`: Either `"ujson"`, the built-in JSON parser, or `"pyarrow"` which dispatches to pyarrow’s `pyarrow.json.read_json`. The `"pyarrow"` is only available when `lines=True`

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'`

Skip to main content

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 [262]: from io import StringIO

In [263]: pd.read_json(StringIO(json))
Out[263]:
   date         B         A
0     1  0.403310  0.176444
1     1  0.301624 -0.154951
2     1 -1.369849 -2.179861
3     1  1.462696 -0.954208
4     1 -0.826591 -1.743161
```

Reading from a file:

```
In [264]: pd.read_json("test.json")
Out[264]:
                   A         B  date  ints  bools
2013-01-01 -0.121306 -0.097883  1356     0   True
2013-01-02  0.695775  0.341734  1356     1   True
2013-01-03  0.959726 -1.110336  1356     2   True
2013-01-04 -0.619976  0.149748  1356     3   True
2013-01-05 -0.732339  0.687738  1356     4   True
```

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

```
In [265]: pd.read_json("test.json", dtype=object).dtypes
Out[265]:
A        object
B        object
date     object
ints     object
```

Skip to main content

```
bools    object
dtype: object
```

Specify dtypes for conversion:

```
In [266]: pd.read_json("test.json", dtype={"A": "float32", "bools": "int8"}).dtypes
Out[266]:
A        float32
B        float64
date       int64
ints       int64
bools       int8
dtype: object
```

Preserve string indices:

```
In [267]: from io import StringIO

In [268]: si = pd.DataFrame(
   .....:     np.zeros((4, 4)), columns=list(range(4)), index=[str(i) for i in range(4)
   .....: )
   .....:

In [269]: si
Out[269]:
     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 [270]: si.index
Out[270]: Index(['0', '1', '2', '3'], dtype='object')

In [271]: si.columns
Out[271]: Index([0, 1, 2, 3], dtype='int64')

In [272]: json = si.to_json()

In [273]: sij = pd.read_json(StringIO(json), convert_axes=False)

In [274]: sij
Out[274]:
   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 [275]: sij.index
Out[275]: Index(['0', '1', '2', '3'], dtype='object')
```

Skip to main content

```
In [276]: sij.columns
Out[276]: Index(['0', '1', '2', '3'], dtype='object')
```

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

```
In [277]: from io import StringIO

In [278]: json = dfj2.to_json(date_unit="ns")

# Try to parse timestamps as milliseconds -> Won't Work
In [279]: dfju = pd.read_json(StringIO(json), date_unit="ms")

In [280]: dfju
Out[280]:
                            A         B        date  ints  bools
1356998400000000000 -0.121306 -0.097883  1356998400     0   True
1357084800000000000  0.695775  0.341734  1356998400     1   True
1357171200000000000  0.959726 -1.110336  1356998400     2   True
1357257600000000000 -0.619976  0.149748  1356998400     3   True
1357344000000000000 -0.732339  0.687738  1356998400     4   True

# Let pandas detect the correct precision
In [281]: dfju = pd.read_json(StringIO(json))

In [282]: dfju
Out[282]:
                   A         B       date  ints  bools
2013-01-01 -0.121306 -0.097883 2013-01-01     0   True
2013-01-02  0.695775  0.341734 2013-01-01     1   True
2013-01-03  0.959726 -1.110336 2013-01-01     2   True
2013-01-04 -0.619976  0.149748 2013-01-01     3   True
2013-01-05 -0.732339  0.687738 2013-01-01     4   True

# Or specify that all timestamps are in nanoseconds
In [283]: dfju = pd.read_json(StringIO(json), date_unit="ns")

In [284]: dfju
Out[284]:
                   A         B        date  ints  bools
2013-01-01 -0.121306 -0.097883  1356998400     0   True
2013-01-02  0.695775  0.341734  1356998400     1   True
2013-01-03  0.959726 -1.110336  1356998400     2   True
2013-01-04 -0.619976  0.149748  1356998400     3   True
2013-01-05 -0.732339  0.687738  1356998400     4   True
```

By setting the `dtype_backend` argument you can control the default dtypes used for the resulting DataFrame.

```
In [285]: data = (
   .....:  '{"a":{"0":1,"1":3},"b":{"0":2.5,"1":4.5},"c":{"0":true,"1":false},"d":{"0":
   .....:  '"e":{"0":null,"1":6.0},"f":{"0":null,"1":7.5},"g":{"0":null,"1":true},"h":{
```

Skip to main content

```
   .....:

In [286]: df = pd.read_json(StringIO(data), dtype_backend="pyarrow")

In [287]: df
Out[287]:
   a    b      c  d     e     f     g     h           i     j
0  1  2.5   True  a  <NA>  <NA>  <NA>  <NA>  12-31-2019  None
1  3  4.5  False  b     6   7.5  True     a  12-31-2019  None

In [288]: df.dtypes
Out[288]:
a     int64[pyarrow]
b    double[pyarrow]
c      bool[pyarrow]
d    string[pyarrow]
e     int64[pyarrow]
f    double[pyarrow]
g      bool[pyarrow]
h    string[pyarrow]
i    string[pyarrow]
j      null[pyarrow]
dtype: object
```

Normalization

data into a flat table.

```
In [289]: data = [
   .....:     {"id": 1, "name": {"first": "Coleen", "last": "Volk"}},
   .....:     {"name": {"given": "Mark", "family": "Regner"}},
   .....:     {"id": 2, "name": "Faye Raker"},
   .....: ]
   .....:

In [290]: pd.json_normalize(data)
Out[290]:
    id name.first name.last name.given name.family        name
0  1.0     Coleen      Volk        NaN         NaN         NaN
1  NaN        NaN       NaN       Mark      Regner         NaN
2  2.0        NaN       NaN        NaN         NaN  Faye Raker
```

```
In [291]: data = [
   .....:     {
   .....:         "state": "Florida",
   .....:         "shortname": "FL",
   .....:         "info": {"governor": "Rick Scott"},
   .....:         "county": [
```

Skip to main content

```
   .....:             {"name": "Palm Beach", "population": 60000},
   .....:         ],
   .....:     },
   .....:     {
   .....:         "state": "Ohio",
   .....:         "shortname": "OH",
   .....:         "info": {"governor": "John Kasich"},
   .....:         "county": [
   .....:             {"name": "Summit", "population": 1234},
   .....:             {"name": "Cuyahoga", "population": 1337},
   .....:         ],
   .....:     },
   .....: ]
   .....:

In [292]: pd.json_normalize(data, "county", ["state", "shortname", ["info", "governor"]
Out[292]:
         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 [293]: data = [
   .....:     {
   .....:         "CreatedBy": {"Name": "User001"},
   .....:         "Lookup": {
   .....:             "TextField": "Some text",
   .....:             "UserField": {"Id": "ID001", "Name": "Name001"},
   .....:         },
   .....:         "Image": {"a": "b"},
   .....:     }
   .....: ]
   .....:

In [294]: pd.json_normalize(data, max_level=1)
Out[294]:
  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.

Skip to main content

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 [295]: from io import StringIO

In [296]: jsonl = """
   .....:     {"a": 1, "b": 2}
   .....:     {"a": 3, "b": 4}
   .....: """
   .....:

In [297]: df = pd.read_json(StringIO(jsonl), lines=True)

In [298]: df
Out[298]:
   a  b
0  1  2
1  3  4

In [299]: df.to_json(orient="records", lines=True)
Out[299]: '{"a":1,"b":2}\n{"a":3,"b":4}\n'

# reader is an iterator that returns ``chunksize`` lines each iteration
In [300]: with pd.read_json(StringIO(jsonl), lines=True, chunksize=1) as reader:
   .....:     reader
   .....:     for chunk in reader:
   .....:         print(chunk)
   .....:
Empty DataFrame
Columns: []
Index: []
   a  b
0  1  2
   a  b
1  3  4
```

Line-limited json can also be read using the pyarrow reader by specifying `engine="pyarrow"`.

```
In [301]: from io import BytesIO

In [302]: df = pd.read_json(BytesIO(jsonl.encode()), lines=True, engine="pyarrow")

In [303]: df
Out[303]:
   a  b
0  1  2
1  3  4
```

> **New in version 2.0.0.**

Skip to main content

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 [304]: 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 [305]: df
Out[305]:
     A  B          C
idx
0    1  a 2016-01-01
1    2  b 2016-01-02
2    3  c 2016-01-03

In [306]: df.to_json(orient="table", date_format="iso")
Out[306]: '{"schema":{"fields":[{"name":"idx","type":"integer"},{"name":"A","type":"int
```

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

Skip to main content

pandas type	Table Schema type
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 [307]: from pandas.io.json import build_table_schema

In [308]: s = pd.Series(pd.date_range("2016", periods=4))

In [309]: build_table_schema(s)
Out[309]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'datetime'}],
 'primaryKey': ['index'],
 'pandas_version': '1.4.0'}
```

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

```
In [310]: s_tz = pd.Series(pd.date_range("2016", periods=12, tz="US/Central"))

In [311]: build_table_schema(s_tz)
Out[311]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'datetime', 'tz': 'US/Central'}],
 'primaryKey': ['index'],
 'pandas_version': '1.4.0'}
```

- 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 [312]: s_per = pd.Series(1, index=pd.period_range("2016", freq="Y-DEC", periods=

In [313]: build_table_schema(s_per)
```

Skip to main content

```
   {'name': 'values', 'type': 'integer'}],
  'primaryKey': ['index'],
  'pandas_version': '1.4.0'}
```

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

```
In [314]: s_cat = pd.Series(pd.Categorical(["a", "b", "a"]))

In [315]: build_table_schema(s_cat)
Out[315]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values',
   'type': 'any',
   'constraints': {'enum': ['a', 'b']},
   'ordered': False}],
 'primaryKey': ['index'],
 'pandas_version': '1.4.0'}
```

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

```
In [316]: s_dupe = pd.Series([1, 2], index=[1, 1])

In [317]: build_table_schema(s_dupe)
Out[317]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
 'pandas_version': '1.4.0'}
```

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

```
In [318]: s_multi = pd.Series(1, index=pd.MultiIndex.from_product([("a", "b"), (0,

In [319]: build_table_schema(s_multi)
Out[319]:
{'fields': [{'name': 'level_0', 'type': 'string'},
  {'name': 'level_1', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
 'primaryKey': FrozenList(['level_0', 'level_1']),
 'pandas_version': '1.4.0'}
```

- 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

Skip to main content

- 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.

`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 [320]: 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 [321]: df
Out[321]:
     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 [322]: df.dtypes
Out[322]:
foo             int64
bar            object
baz    datetime64[ns]
qux          category
dtype: object

In [323]: df.to_json("test.json", orient="table")

In [324]: new_df = pd.read_json("test.json", orient="table")

In [325]: new_df
Out[325]:
     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 [326]: new_df.dtypes
Out[326]:
foo             int64
```

Skip to main content

```
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 [327]: df.index.name = "index"

In [328]: df.to_json("test.json", orient="table")

In [329]: new_df = pd.read_json("test.json", orient="table")

In [330]: print(new_df.index.name)
None
```

When using `orient='table'` along with user-defined `ExtensionArray`, the generated schema will contain an additional `extDtype` key in the respective `fields` element. This extra key is not standard but does enable JSON roundtrips for extension types (e.g. `read_json(df.to_json(orient="table"), orient="table")`).

The `extDtype` key carries the name of the extension, if you have properly registered the `ExtensionDtype`, pandas will use said name to perform a lookup into the registry and re-convert the serialized data into your custom dtype.

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.

Skip to main content

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 [320]: url = "https://www.fdic.gov/resources/resolutions/bank-failures/failed-bank-l
In [321]: pd.read_html(url)
Out[321]:
[                        Bank NameBank           CityCity StateSt  ...             Ac
 0                   Almena State Bank             Almena      KS  ...
 1           First City Bank of Florida  Fort Walton Beach      FL  ...            Unit
 2                 The First State Bank      Barboursville      WV  ...
 3                   Ericson State Bank            Ericson      NE  ...           Farme
 4     City National Bank of New Jersey             Newark      NJ  ...
 ..                                 ...                ...     ...  ...
 558                 Superior Bank, FSB           Hinsdale      IL  ...
 559                Malta National Bank              Malta      OH  ...
 560    First Alliance Bank & Trust Co.         Manchester      NH  ...  Southern New H
 561  National State Bank of Metropolis         Metropolis      IL  ...             Ba
 562                   Bank of Honolulu           Honolulu      HI  ...

 [563 rows x 7 columns]]
```

Note

The data from the above URL changes every Monday so the resulting data above may be slightly different.

Read a URL while passing headers alongside the HTTP request:

```
In [322]: url = 'https://www.sump.org/notes/request/' # HTTP request reflector
In [323]: pd.read_html(url)
Out[323]:
[                  0                     1
 0     Remote Socket:  51.15.105.256:51760
 1  Protocol Version:              HTTP/1.1
 2    Request Method:                   GET
 3       Request URI:       /notes/request/
 4     Request Query:                   NaN,
 0   Accept-Encoding:              identity
 1              Host:          www.sump.org
 2        User-Agent:     Python-urllib/3.8
 3        Connection:                 close]
In [324]: headers = {
```

Skip to main content

```
In [327]:     'Connection':'keep-alive',
In [328]:     'Auth':'Bearer 2*/f3+fe68df*4'
In [329]: }
In [340]: pd.read_html(url, storage_options=headers)
Out[340]:
[                  0                    1
 0     Remote Socket:  51.15.105.256:51760
 1  Protocol Version:             HTTP/1.1
 2    Request Method:                  GET
 3       Request URI:      /notes/request/
 4     Request Query:                  NaN,
 0        User-Agent: Mozilla Firefox v14.0
 1    AcceptEncoding:   gzip,  deflate,  br
 2            Accept:      application/json
 3        Connection:             keep-alive
 4              Auth:  Bearer 2*/f3+fe68df*4]
```

Note

We see above that the headers we passed are reflected in the HTTP request.

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

```
In [331]: html_str = """
   .....:           <table>
   .....:               <tr>
   .....:                   <th>A</th>
   .....:                   <th colspan="1">B</th>
   .....:                   <th rowspan="1">C</th>
   .....:               </tr>
   .....:               <tr>
   .....:                   <td>a</td>
   .....:                   <td>b</td>
   .....:                   <td>c</td>
   .....:               </tr>
   .....:           </table>
   .....:       """
   .....:

In [332]: with open("tmp.html", "w") as f:
   .....:     f.write(html_str)
   .....:

In [333]: df = pd.read_html("tmp.html")

In [334]: df[0]
Out[334]:
   A  B  C
0  a  b  c
```

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

Skip to main content

```
In [335]: dfs = pd.read_html(StringIO(html_str))

In [336]: dfs[0]
Out[336]:
   A  B  C
0  a  b  c
```

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 (`range` works as well):

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

Skip to main content

```
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?oldid=899173761"
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"])
```

Skip to main content

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"])
```

Links can be extracted from cells along with the text using `extract_links="all"`.

```
In [337]: html_table = """
   .....: <table>
   .....:   <tr>
   .....:     <th>GitHub</th>
   .....:   </tr>
   .....:   <tr>
   .....:     <td><a href="https://github.com/pandas-dev/pandas">pandas</a></td>
   .....:   </tr>
   .....: </table>
   .....: """
   .....:

In [338]: df = pd.read_html(
   .....:     StringIO(html_table),
   .....:     extract_links="all"
   .....: )[0]
   .....:

In [339]: df
Out[339]:
                                    (GitHub, None)
0  (pandas, https://github.com/pandas-dev/pandas)

In [340]: df[("GitHub", None)]
Out[340]:
0    (pandas, https://github.com/pandas-dev/pandas)
Name: (GitHub, None), dtype: object

In [341]: df[("GitHub", None)].str[1]
Out[341]:
0    https://github.com/pandas-dev/pandas
Name: (GitHub, None), dtype: object
```

> **New in version 1.5.0.**

Skip to main content

`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 `DataFrame.to_html()` for the full set of options.

Note

In an HTML-rendering supported environment like a Jupyter Notebook, `display(HTML(...))` will render the raw HTML into the environment.

```
In [342]: from IPython.display import display, HTML

In [343]: df = pd.DataFrame(np.random.randn(2, 2))

In [344]: df
Out[344]:
          0         1
0 -0.345352  1.314232
1  0.690579  0.995761

In [345]: html = df.to_html()

In [346]: print(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>-0.345352</td>
      <td>1.314232</td>
    </tr>
    <tr>
      <th>1</th>
      <td>0.690579</td>
      <td>0.995761</td>
    </tr>
  </tbody>
</table>
```

Skip to main content

```
In [347]: display(HTML(html))
<IPython.core.display.HTML object>
```

The `columns` argument will limit the columns shown:

```
In [348]: html = df.to_html(columns=[0])

In [349]: print(html)
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.345352</td>
    </tr>
    <tr>
      <th>1</th>
      <td>0.690579</td>
    </tr>
  </tbody>
</table>

In [350]: display(HTML(html))
<IPython.core.display.HTML object>
```

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

```
In [351]: html = df.to_html(float_format="{0:.10f}".format)

In [352]: print(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>-0.3453521949</td>
      <td>1.3142323796</td>
    </tr>
    <tr>
      <th>1</th>
      <td>0.6905793352</td>
      <td>0.9957609037</td>
```

Skip to main content

```
</table>

In [353]: display(HTML(html))
<IPython.core.display.HTML object>
```

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

```
In [354]: html = df.to_html(bold_rows=False)

In [355]: print(html)
<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>-0.345352</td>
      <td>1.314232</td>
    </tr>
    <tr>
      <td>1</td>
      <td>0.690579</td>
      <td>0.995761</td>
    </tr>
  </tbody>
</table>

In [356]: display(HTML(html))
<IPython.core.display.HTML object>
```

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 [357]: 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>-0.345352</td>
      <td>1.314232</td>
```

Skip to main content

```
      <th>1</th>
      <td>0.690579</td>
      <td>0.995761</td>
    </tr>
  </tbody>
</table>
```

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

```
In [358]: url_df = pd.DataFrame(
   .....:     {
   .....:         "name": ["Python", "pandas"],
   .....:         "url": ["https://www.python.org/", "https://pandas.pydata.org"],
   .....:     }
   ..... : )
   .....:

In [359]: html = url_df.to_html(render_links=True)

In [360]: print(html)
<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>
      <th>1</th>
      <td>pandas</td>
      <td><a href="https://pandas.pydata.org" target="_blank">https://pandas.pydata.org</a></td>
    </tr>
  </tbody>
</table>

In [361]: display(HTML(html))
<IPython.core.display.HTML object>
```

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

Skip to main content

Escaped:

```
In [363]: html = df.to_html()

In [364]: print(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;amp;</td>
      <td>2.396780</td>
    </tr>
    <tr>
      <th>1</th>
      <td>&amp;lt;</td>
      <td>0.014871</td>
    </tr>
    <tr>
      <th>2</th>
      <td>&amp;gt;</td>
      <td>3.357427</td>
    </tr>
  </tbody>
</table>

In [365]: display(HTML(html))
<IPython.core.display.HTML object>
```

Not escaped:

```
In [366]: html = df.to_html(escape=False)

In [367]: print(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>2.396780</td>
    </tr>
```

Skip to main content

```
      <th>1</th>
      <td><</td>
      <td>0.014871</td>
    </tr>
    <tr>
      <th>2</th>
      <td>></td>
      <td>3.357427</td>
    </tr>
  </tbody>
</table>

In [368]: display(HTML(html))
<IPython.core.display.HTML object>
```

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.

Skip to main content

- 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.

- 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.

LaTeX

> ***New in version 1.3.0.***

Currently there are no methods to read from LaTeX, only output methods.

Writing to LaTeX files

Skip to main content

Note

DataFrame *and* Styler objects currently have a `to_latex` method. We recommend using the Styler.to_latex() method over DataFrame.to_latex() due to the former's greater flexibility with conditional styling, and the latter's possible future deprecation.

Review the documentation for Styler.to_latex, which gives examples of conditional styling and explains the operation of its keyword arguments.

For simple application the following pattern is sufficient.

```
In [369]: df = pd.DataFrame([[1, 2], [3, 4]], index=["a", "b"], columns=["c", "d"])

In [370]: print(df.style.to_latex())
\begin{tabular}{lrr}
 & c & d \\
a & 1 & 2 \\
b & 3 & 4 \\
\end{tabular}
```

To format values before output, chain the Styler.format method.

```
In [371]: print(df.style.format("€ {}").to_latex())
\begin{tabular}{lrr}
 & c & d \\
a & € 1 & € 2 \\
b & € 3 & € 4 \\
\end{tabular}
```

XML

Reading XML

New in version 1.3.0.

The top-level `read_xml()` function can accept an XML string/file/URL and will parse nodes and attributes into a pandas `DataFrame`.

Skip to main content

Note

Since there is no standard XML structure where design types can vary in many ways, `read_xml` works best with flatter, shallow versions. If an XML document is deeply nested, use the `stylesheet` feature to transform XML into a flatter version.

Let's look at a few examples.

Read an XML string:

```
In [372]: from io import StringIO

In [373]: xml = """<?xml version="1.0" encoding="UTF-8"?>
   .....: <bookstore>
   .....:   <book category="cooking">
   .....:     <title lang="en">Everyday Italian</title>
   .....:     <author>Giada De Laurentiis</author>
   .....:     <year>2005</year>
   .....:     <price>30.00</price>
   .....:   </book>
   .....:   <book category="children">
   .....:     <title lang="en">Harry Potter</title>
   .....:     <author>J K. Rowling</author>
   .....:     <year>2005</year>
   .....:     <price>29.99</price>
   .....:   </book>
   .....:   <book category="web">
   .....:     <title lang="en">Learning XML</title>
   .....:     <author>Erik T. Ray</author>
   .....:     <year>2003</year>
   .....:     <price>39.95</price>
   .....:   </book>
   .....: </bookstore>"""
   .....:

In [374]: df = pd.read_xml(StringIO(xml))

In [375]: df
Out[375]:
   category             title               author  year  price
0   cooking  Everyday Italian  Giada De Laurentiis  2005  30.00
1  children      Harry Potter         J K. Rowling  2005  29.99
2       web      Learning XML          Erik T. Ray  2003  39.95
```

Read a URL with no options:

```
In [376]: df = pd.read_xml("https://www.w3schools.com/xml/books.xml")

In [377]: df
Out[377]:
```

Skip to main content

```
1  children       Harry Potter            J K. Rowling  2005  29.99       None
2       web  XQuery Kick Start  Vaidyanathan Nagarajan  2003  49.99       None
3       web       Learning XML             Erik T. Ray  2003  39.95  paperback
```

Read in the content of the "books.xml" file and pass it to `read_xml` as a string:

```
In [378]: file_path = "books.xml"

In [379]: with open(file_path, "w") as f:
   .....:     f.write(xml)
   .....:

In [380]: with open(file_path, "r") as f:
   .....:     df = pd.read_xml(StringIO(f.read()))
   .....:

In [381]: df
Out[381]:
   category             title              author  year  price
0   cooking  Everyday Italian  Giada De Laurentiis  2005  30.00
1  children      Harry Potter         J K. Rowling  2005  29.99
2       web      Learning XML          Erik T. Ray  2003  39.95
```

Read in the content of the "books.xml" as instance of `StringIO` or `BytesIO` and pass it to `read_xml`:

```
In [382]: with open(file_path, "r") as f:
   .....:     sio = StringIO(f.read())
   .....:

In [383]: df = pd.read_xml(sio)

In [384]: df
Out[384]:
   category             title              author  year  price
0   cooking  Everyday Italian  Giada De Laurentiis  2005  30.00
1  children      Harry Potter         J K. Rowling  2005  29.99
2       web      Learning XML          Erik T. Ray  2003  39.95
```

```
In [385]: with open(file_path, "rb") as f:
   .....:     bio = BytesIO(f.read())
   .....:

In [386]: df = pd.read_xml(bio)

In [387]: df
Out[387]:
   category             title              author  year  price
0   cooking  Everyday Italian  Giada De Laurentiis  2005  30.00
1  children      Harry Potter         J K. Rowling  2005  29.99
```

Skip to main content

Even read XML from AWS S3 buckets such as NIH NCBI PMC Article Datasets providing Biomedical and Life Science Jorurnals:

```
>>> df = pd.read_xml(
...     "s3://pmc-oa-opendata/oa_comm/xml/all/PMC1236943.xml",
...     xpath=".//journal-meta",
...)
>>> df.head(1)
      journal-id  journal-title  issn  publisher
0 Cardiovasc Ultrasound Cardiovascular Ultrasound 1476-7120 NaN
```

With lxml as default `parser`, you access the full-featured XML library that extends Python's ElementTree API. One powerful tool is ability to query nodes selectively or conditionally with more expressive XPath:

```
In [388]: df = pd.read_xml(file_path, xpath="//book[year=2005]")

In [389]: df
Out[389]:
   category             title              author  year  price
0   cooking  Everyday Italian  Giada De Laurentiis  2005  30.00
1  children      Harry Potter         J K. Rowling  2005  29.99
```

Specify only elements or only attributes to parse:

```
In [390]: df = pd.read_xml(file_path, elems_only=True)

In [391]: df
Out[391]:
              title              author  year  price
0  Everyday Italian  Giada De Laurentiis  2005  30.00
1      Harry Potter         J K. Rowling  2005  29.99
2      Learning XML          Erik T. Ray  2003  39.95
```

```
In [392]: df = pd.read_xml(file_path, attrs_only=True)

In [393]: df
Out[393]:
   category
0   cooking
1  children
2       web
```

XML documents can have namespaces with prefixes and default namespaces without prefixes both of which are denoted with a special attribute `xmlns`. In order to parse by node under a namespace context, `xpath` must reference a prefix.

Skip to main content

For example, below XML contains a namespace with prefix, `doc`, and URI at `https://example.com`. In order to parse `doc:row` nodes, `namespaces` must be used.

```
In [394]: xml = """<?xml version='1.0' encoding='utf-8'?>
   .....: <doc:data xmlns:doc="https://example.com">
   .....:   <doc:row>
   .....:     <doc:shape>square</doc:shape>
   .....:     <doc:degrees>360</doc:degrees>
   .....:     <doc:sides>4.0</doc:sides>
   .....:   </doc:row>
   .....:   <doc:row>
   .....:     <doc:shape>circle</doc:shape>
   .....:     <doc:degrees>360</doc:degrees>
   .....:     <doc:sides/>
   .....:   </doc:row>
   .....:   <doc:row>
   .....:     <doc:shape>triangle</doc:shape>
   .....:     <doc:degrees>180</doc:degrees>
   .....:     <doc:sides>3.0</doc:sides>
   .....:   </doc:row>
   .....: </doc:data>"""
   .....:

In [395]: df = pd.read_xml(StringIO(xml),
   .....:                  xpath="//doc:row",
   .....:                  namespaces={"doc": "https://example.com"})
   .....:

In [396]: df
Out[396]:
      shape  degrees  sides
0    square      360    4.0
1    circle      360    NaN
2  triangle      180    3.0
```

Similarly, an XML document can have a default namespace without prefix. Failing to assign a temporary prefix will return no nodes and raise a `ValueError`. But assigning *any* temporary name to correct URI allows parsing by nodes.

```
In [397]: xml = """<?xml version='1.0' encoding='utf-8'?>
   .....: <data xmlns="https://example.com">
   .....:  <row>
   .....:    <shape>square</shape>
   .....:    <degrees>360</degrees>
   .....:    <sides>4.0</sides>
   .....:  </row>
   .....:  <row>
   .....:    <shape>circle</shape>
   .....:    <degrees>360</degrees>
   .....:    <sides/>
   .....:  </row>
   .....:  <row>
```

Skip to main content

```
   .....:    <sides>3.0</sides>
   .....:  </row>
   .....: </data>"""
   .....:

In [398]: df = pd.read_xml(StringIO(xml),
   .....:                  xpath="//pandas:row",
   .....:                  namespaces={"pandas": "https://example.com"})
   .....:

In [399]: df
Out[399]:
      shape  degrees  sides
0    square      360    4.0
1    circle      360    NaN
2  triangle      180    3.0
```

However, if XPath does not reference node names such as default, `/*`, then `namespaces` is not required.

Skip to main content

Note

Since `xpath` identifies the parent of content to be parsed, only immediate desendants which include child nodes or current attributes are parsed. Therefore, `read_xml` will not parse the text of grandchildren or other descendants and will not parse attributes of any descendant. To retrieve lower level content, adjust xpath to lower level. For example,

```
In [400]: xml = """
   .....: <data>
   .....:   <row>
   .....:     <shape sides="4">square</shape>
   .....:     <degrees>360</degrees>
   .....:   </row>
   .....:   <row>
   .....:     <shape sides="0">circle</shape>
   .....:     <degrees>360</degrees>
   .....:   </row>
   .....:   <row>
   .....:     <shape sides="3">triangle</shape>
   .....:     <degrees>180</degrees>
   .....:   </row>
   .....: </data>"""
   .....:

In [401]: df = pd.read_xml(StringIO(xml), xpath="./row")

In [402]: df
Out[402]:
      shape  degrees
0    square      360
1    circle      360
2  triangle      180
```

shows the attribute `sides` on `shape` element was not parsed as expected since this attribute resides on the child of `row` element and not `row` element itself. In other words, `sides` attribute is a grandchild level descendant of `row` element. However, the `xpath` targets `row` element which covers only its children and attributes.

With lxml as parser, you can flatten nested XML documents with an XSLT script which also can be string/file/URL types. As background, XSLT is a special-purpose language written in a special XML file that can transform original XML documents into other XML, HTML, even text (CSV, JSON, etc.) using an XSLT processor.

For example, consider this somewhat nested structure of Chicago “L” Rides where station and rides elements encapsulate data in their own sections. With below XSLT, `lxml` can transform

Skip to main content

original nested document into a flatter output (as shown below for demonstration) for easier parse into `DataFrame`:

```
In [403]: xml = """<?xml version='1.0' encoding='utf-8'?>
   .....:  <response>
   .....:   <row>
   .....:     <station id="40850" name="Library"/>
   .....:     <month>2020-09-01T00:00:00</month>
   .....:     <rides>
   .....:       <avg_weekday_rides>864.2</avg_weekday_rides>
   .....:       <avg_saturday_rides>534</avg_saturday_rides>
   .....:       <avg_sunday_holiday_rides>417.2</avg_sunday_holiday_rides>
   .....:     </rides>
   .....:   </row>
   .....:   <row>
   .....:     <station id="41700" name="Washington/Wabash"/>
   .....:     <month>2020-09-01T00:00:00</month>
   .....:     <rides>
   .....:       <avg_weekday_rides>2707.4</avg_weekday_rides>
   .....:       <avg_saturday_rides>1909.8</avg_saturday_rides>
   .....:       <avg_sunday_holiday_rides>1438.6</avg_sunday_holiday_rides>
   .....:     </rides>
   .....:   </row>
   .....:   <row>
   .....:     <station id="40380" name="Clark/Lake"/>
   .....:     <month>2020-09-01T00:00:00</month>
   .....:     <rides>
   .....:       <avg_weekday_rides>2949.6</avg_weekday_rides>
   .....:       <avg_saturday_rides>1657</avg_saturday_rides>
   .....:       <avg_sunday_holiday_rides>1453.8</avg_sunday_holiday_rides>
   .....:     </rides>
   .....:   </row>
   .....:  </response>"""
   .....:

In [404]: xsl = """<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
   .....:    <xsl:output method="xml" omit-xml-declaration="no" indent="yes"/>
   .....:    <xsl:strip-space elements="*"/>
   .....:    <xsl:template match="/response">
   .....:       <xsl:copy>
   .....:         <xsl:apply-templates select="row"/>
   .....:       </xsl:copy>
   .....:    </xsl:template>
   .....:    <xsl:template match="row">
   .....:       <xsl:copy>
   .....:         <station_id><xsl:value-of select="station/@id"/></station_id>
   .....:         <station_name><xsl:value-of select="station/@name"/></station_name>
   .....:         <xsl:copy-of select="month|rides/*"/>
   .....:       </xsl:copy>
   .....:    </xsl:template>
   .....:  </xsl:stylesheet>"""
   .....:

In [405]: output = """<?xml version='1.0' encoding='utf-8'?>
   .....:  <response>
```

Skip to main content

```
    .....:       <station_name>Library</station_name>
    .....:       <month>2020-09-01T00:00:00</month>
    .....:       <avg_weekday_rides>864.2</avg_weekday_rides>
    .....:       <avg_saturday_rides>534</avg_saturday_rides>
    .....:       <avg_sunday_holiday_rides>417.2</avg_sunday_holiday_rides>
    .....:    </row>
    .....:    <row>
    .....:       <station_id>41700</station_id>
    .....:       <station_name>Washington/Wabash</station_name>
    .....:       <month>2020-09-01T00:00:00</month>
    .....:       <avg_weekday_rides>2707.4</avg_weekday_rides>
    .....:       <avg_saturday_rides>1909.8</avg_saturday_rides>
    .....:       <avg_sunday_holiday_rides>1438.6</avg_sunday_holiday_rides>
    .....:    </row>
    .....:    <row>
    .....:       <station_id>40380</station_id>
    .....:       <station_name>Clark/Lake</station_name>
    .....:       <month>2020-09-01T00:00:00</month>
    .....:       <avg_weekday_rides>2949.6</avg_weekday_rides>
    .....:       <avg_saturday_rides>1657</avg_saturday_rides>
    .....:       <avg_sunday_holiday_rides>1453.8</avg_sunday_holiday_rides>
    .....:    </row>
    .....: </response>"""
    .....:

In [406]: df = pd.read_xml(StringIO(xml), stylesheet=xsl)

In [407]: df
Out[407]:
   station_id       station_name                month  avg_weekday_rides  avg_saturday_
0       40850            Library  2020-09-01T00:00:00              864.2
1       41700  Washington/Wabash  2020-09-01T00:00:00             2707.4             1
2       40380        Clark/Lake  2020-09-01T00:00:00             2949.6             1
```

For very large XML files that can range in hundreds of megabytes to gigabytes, `pandas.read_xml()` supports parsing such sizeable files using lxml's iterparse and etree's iterparse which are memory-efficient methods to iterate through an XML tree and extract specific elements and attributes. without holding entire tree in memory.

> ***New in version 1.5.0.***

To use this feature, you must pass a physical XML file path into `read_xml` and use the `iterparse` argument. Files should not be compressed or point to online sources but stored on local disk. Also, `iterparse` should be a dictionary where the key is the repeating nodes in document (which become the rows) and the value is a list of any element or attribute that is a descendant (i.e., child, grandchild) of repeating node. Since XPath is not used in this method, descendants do not need to share same relationship with one another. Below shows example of reading in Wikipedia's very large (12 GB+) latest article data dump

Skip to main content

```
In [1]: df = pd.read_xml(
...         "/path/to/downloaded/enwikisource-latest-pages-articles.xml",
...         iterparse = {"page": ["title", "ns", "id"]}
...     )
...     df
Out[2]:
                                                    title   ns        id
0                                     Gettysburg Address    0     21450
1                                              Main Page    0     42950
2                            Declaration by United Nations    0      8435
3              Constitution of the United States of America    0      8435
4                      Declaration of Independence (Israel)    0     17858
...                                                     ...  ...       ...
3578760               Page:Black cat 1897 07 v2 n10.pdf/17  104    219649
3578761               Page:Black cat 1897 07 v2 n10.pdf/43  104    219649
3578762               Page:Black cat 1897 07 v2 n10.pdf/44  104    219649
3578763      The History of Tom Jones, a Foundling/Book IX    0  12084291
3578764  Page:Shakespeare of Stratford (1926) Yale.djvu/91  104     21450

[3578765 rows x 3 columns]
```

Writing XML

> **New in version 1.3.0.**

`DataFrame` objects have an instance method `to_xml` which renders the contents of the `DataFrame` as an XML document.

> **Note**
>
> This method does not support special properties of XML including DTD, CData, XSD schemas, processing instructions, comments, and others. Only namespaces at the root level is supported. However, `stylesheet` allows design changes after initial output.

Let's look at a few examples.

Write an XML without options:

```
In [408]: geom_df = pd.DataFrame(
   .....:     {
   .....:         "shape": ["square", "circle", "triangle"],
   .....:         "degrees": [360, 360, 180],
   .....:         "sides": [4, np.nan, 3],
   .....:     }
```

Skip to main content

```
In [409]: print(geom_df.to_xml())
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row>
    <index>0</index>
    <shape>square</shape>
    <degrees>360</degrees>
    <sides>4.0</sides>
  </row>
  <row>
    <index>1</index>
    <shape>circle</shape>
    <degrees>360</degrees>
    <sides/>
  </row>
  <row>
    <index>2</index>
    <shape>triangle</shape>
    <degrees>180</degrees>
    <sides>3.0</sides>
  </row>
</data>
```

Write an XML with new root and row name:

```
In [410]: print(geom_df.to_xml(root_name="geometry", row_name="objects"))
<?xml version='1.0' encoding='utf-8'?>
<geometry>
  <objects>
    <index>0</index>
    <shape>square</shape>
    <degrees>360</degrees>
    <sides>4.0</sides>
  </objects>
  <objects>
    <index>1</index>
    <shape>circle</shape>
    <degrees>360</degrees>
    <sides/>
  </objects>
  <objects>
    <index>2</index>
    <shape>triangle</shape>
    <degrees>180</degrees>
    <sides>3.0</sides>
  </objects>
</geometry>
```

Write an attribute-centric XML:

```
In [411]: print(geom_df.to_xml(attr_cols=geom_df.columns.tolist()))
<?xml version='1.0' encoding='utf-8'?>
```

Skip to main content

```
  <row index="1" shape="circle" degrees="360"/>
  <row index="2" shape="triangle" degrees="180" sides="3.0"/>
</data>
```

Write a mix of elements and attributes:

```
In [412]: print(
   .....:     geom_df.to_xml(
   .....:         index=False,
   .....:         attr_cols=['shape'],
   .....:         elem_cols=['degrees', 'sides'])
   .....: )
   .....:
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row shape="square">
    <degrees>360</degrees>
    <sides>4.0</sides>
  </row>
  <row shape="circle">
    <degrees>360</degrees>
    <sides/>
  </row>
  <row shape="triangle">
    <degrees>180</degrees>
    <sides>3.0</sides>
  </row>
</data>
```

Any `DataFrames` with hierarchical columns will be flattened for XML element names with levels delimited by underscores:

```
In [413]: ext_geom_df = pd.DataFrame(
   .....:     {
   .....:         "type": ["polygon", "other", "polygon"],
   .....:         "shape": ["square", "circle", "triangle"],
   .....:         "degrees": [360, 360, 180],
   .....:         "sides": [4, np.nan, 3],
   .....:     }
   .....: )
   .....:

In [414]: pvt_df = ext_geom_df.pivot_table(index='shape',
   .....:                                  columns='type',
   .....:                                  values=['degrees', 'sides'],
   .....:                                  aggfunc='sum')
   .....:

In [415]: pvt_df
Out[415]:
         degrees         sides
type       other polygon other polygon
```

Skip to main content

```
square       NaN   360.0   NaN     4.0
triangle     NaN   180.0   NaN     3.0

In [416]: print(pvt_df.to_xml())
<?xml version='1.0' encoding='utf-8'?>
<data>
  <row>
    <shape>circle</shape>
    <degrees_other>360.0</degrees_other>
    <degrees_polygon/>
    <sides_other>0.0</sides_other>
    <sides_polygon/>
  </row>
  <row>
    <shape>square</shape>
    <degrees_other/>
    <degrees_polygon>360.0</degrees_polygon>
    <sides_other/>
    <sides_polygon>4.0</sides_polygon>
  </row>
  <row>
    <shape>triangle</shape>
    <degrees_other/>
    <degrees_polygon>180.0</degrees_polygon>
    <sides_other/>
    <sides_polygon>3.0</sides_polygon>
  </row>
</data>
```

Write an XML with default namespace:

```
In [417]: print(geom_df.to_xml(namespaces={"": "https://example.com"}))
<?xml version='1.0' encoding='utf-8'?>
<data xmlns="https://example.com">
  <row>
    <index>0</index>
    <shape>square</shape>
    <degrees>360</degrees>
    <sides>4.0</sides>
  </row>
  <row>
    <index>1</index>
    <shape>circle</shape>
    <degrees>360</degrees>
    <sides/>
  </row>
  <row>
    <index>2</index>
    <shape>triangle</shape>
    <degrees>180</degrees>
    <sides>3.0</sides>
  </row>
</data>
```

Skip to main content

```
In [418]: print(
   .....:     geom_df.to_xml(namespaces={"doc": "https://example.com"},
   .....:                    prefix="doc")
   .....: )
   .....:
<?xml version='1.0' encoding='utf-8'?>
<doc:data xmlns:doc="https://example.com">
  <doc:row>
    <doc:index>0</doc:index>
    <doc:shape>square</doc:shape>
    <doc:degrees>360</doc:degrees>
    <doc:sides>4.0</doc:sides>
  </doc:row>
  <doc:row>
    <doc:index>1</doc:index>
    <doc:shape>circle</doc:shape>
    <doc:degrees>360</doc:degrees>
    <doc:sides/>
  </doc:row>
  <doc:row>
    <doc:index>2</doc:index>
    <doc:shape>triangle</doc:shape>
    <doc:degrees>180</doc:degrees>
    <doc:sides>3.0</doc:sides>
  </doc:row>
</doc:data>
```

Write an XML without declaration or pretty print:

```
In [419]: print(
   .....:     geom_df.to_xml(xml_declaration=False,
   .....:                    pretty_print=False)
   .....: )
   .....:
<data><row><index>0</index><shape>square</shape><degrees>360</degrees><sides>4.0</sides
```

Write an XML and transform with stylesheet:

```
In [420]: xsl = """<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
   .....:     <xsl:output method="xml" omit-xml-declaration="no" indent="yes"/>
   .....:     <xsl:strip-space elements="*"/>
   .....:     <xsl:template match="/data">
   .....:       <geometry>
   .....:         <xsl:apply-templates select="row"/>
   .....:       </geometry>
   .....:     </xsl:template>
   .....:     <xsl:template match="row">
   .....:       <object index="{index}">
   .....:         <xsl:if test="shape!='circle'">
   .....:             <xsl:attribute name="type">polygon</xsl:attribute>
   .....:         </xsl:if>
```

Skip to main content

```
    .....:             <xsl:copy-of select="degrees|sides"/>
    .....:           </property>
    .....:         </object>
    .....:       </xsl:template>
    .....:   </xsl:stylesheet>"""
    .....:

In [421]: print(geom_df.to_xml(stylesheet=xsl))
<?xml version="1.0"?>
<geometry>
  <object index="0" type="polygon">
    <shape>square</shape>
    <property>
      <degrees>360</degrees>
      <sides>4.0</sides>
    </property>
  </object>
  <object index="1">
    <shape>circle</shape>
    <property>
      <degrees>360</degrees>
      <sides/>
    </property>
  </object>
  <object index="2" type="polygon">
    <shape>triangle</shape>
    <property>
      <degrees>180</degrees>
      <sides>3.0</sides>
    </property>
  </object>
</geometry>
```

XML Final Notes

- All XML documents adhere to W3C specifications. Both `etree` and `lxml` parsers will fail to parse any markup document that is not well-formed or follows XML syntax rules. Do be aware HTML is not an XML document unless it follows XHTML specs. However, other popular markup types including KML, XAML, RSS, MusicML, MathML are compliant XML schemas.
- For above reason, if your application builds XML prior to pandas operations, use appropriate DOM libraries like `etree` and `lxml` to build the necessary document and not by string rules.
- With very large XML files (several hundred MBs to GBs), XPath and XSLT can become memory-intensive operations. Be sure to have enough available RAM for reading and writing to large XML files (roughly about 5 times the size of text).
- Because XSLT is a programming language, use it with caution since such scripts can pose a

Skip to main content

test scripts on small fragments before full run.

- The etree parser supports all functionality of both `read_xml` and `to_xml` except for complex XPath and any XSLT. Though limited in features, `etree` is still a reliable and capable parser and tree builder. Its performance may trail `lxml` to a certain degree for larger files but relatively unnoticeable on small to medium size files.

Excel files

The `read_excel()` method can read Excel 2007+ (`.xlsx`) files using the `openpyxl` Python module. Excel 2003 (`.xls`) files can be read using `xlrd`. Binary Excel (`.xlsb`) files can be read using `pyxlsb`. All formats can be read using calamine engine. 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.

Note

When `engine=None`, the following logic will be used to determine the engine:

- If `path_or_buffer` is an OpenDocument format (.odf, .ods, .odt), then odf will be used.
- Otherwise if `path_or_buffer` is an xls format, `xlrd` will be used.
- Otherwise if `path_or_buffer` is in xlsb format, `pyxlsb` will be used.
- Otherwise `openpyxl` will be used.

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.

When using the `engine_kwargs` parameter, pandas will pass these arguments to the engine. For this, it is important to know which function pandas is using internally.

- For the engine openpyxl, pandas is using `openpyxl.load_workbook()` to read in (`.xlsx`) and (`.xlsm`) files.
- For the engine xlrd, pandas is using `xlrd.open_workbook()` to read in (`.xls`) files.
- For the engine pyxlsb, pandas is using `pyxlsb.open_workbook()` to read in (`.xlsb`) files.
- For the engine odf, pandas is using `odf.opendocument.load()` to read in (`.ods`) files.

Skip to main content

- For the engine calamine, pandas is using `python_calamine.load_workbook()` to read in (`.xlsx`), (`.xlsm`), (`.xls`), (`.xlsb`), (`.ods`) files.

```
# 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(
```

Skip to main content

```
    "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
```

Skip to main content

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

In [423]: df.to_excel("path_to_file.xlsx")

In [424]: df = pd.read_excel("path_to_file.xlsx", index_col=[0, 1])

In [425]: df
Out[425]:
     a  b
a c  1  5
  d  2  6
```

Skip to main content

```
b c  3  7
  d  4  8
```

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

```
In [426]: df.index = df.index.set_names(["lvl1", "lvl2"])

In [427]: df.to_excel("path_to_file.xlsx")

In [428]: df = pd.read_excel("path_to_file.xlsx", index_col=[0, 1])

In [429]: df
Out[429]:
           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`:

```
In [430]: df.columns = pd.MultiIndex.from_product([["a"], ["b", "d"]], names=["c1", "c2

In [431]: df.to_excel("path_to_file.xlsx")

In [432]: df = pd.read_excel("path_to_file.xlsx", index_col=[0, 1], header=[0, 1])

In [433]: df
Out[433]:
c1         a
c2         b  d
lvl1 lvl2
a    c     1  5
     d     2  6
b    c     3  7
     d     4  8
```

Missing values in columns specified in `index_col` will be forward filled to allow roundtripping with `to_excel` for `merged_cells=True`. To avoid forward filling the missing values use `set_index` after reading the data instead of `index_col`.

Parsing specific columns

It is often the case that users will insert columns to do temporary computations in Excel and you

Skip to main content

specify a subset of columns to parse.

You can specify a comma-delimited set of Excel columns and ranges as a string:

```
pd.read_excel("path_to_file.xls", "Sheet1", usecols="A,C:E")
```

If `usecols` is a list of integers, then it is assumed to be the file column indices to be parsed.

```
pd.read_excel("path_to_file.xls", "Sheet1", usecols=[0, 2, 3])
```

Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`.

If `usecols` is a list of strings, it is assumed that each string corresponds to a column name provided either by the user in `names` or inferred from the document header row(s). Those strings define which columns will be parsed:

```
pd.read_excel("path_to_file.xls", "Sheet1", usecols=["foo", "bar"])
```

Element order is ignored, so `usecols=['baz', 'joe']` is the same as `['joe', 'baz']`.

If `usecols` is callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to `True`.

```
pd.read_excel("path_to_file.xls", "Sheet1", usecols=lambda x: x.isalpha())
```

Parsing dates

Datetime-like values are normally automatically converted to the appropriate dtype when reading the excel file. But if you have a column of strings that *look* like dates (but are not actually formatted as dates in excel), you can use the `parse_dates` keyword to parse those strings to datetimes:

```
pd.read_excel("path_to_file.xls", "Sheet1", parse_dates=["date_strings"])
```

Cell converters

Skip to main content

It is possible to transform the contents of Excel cells via the `converters` option. For instance, to convert a column to boolean:

```
pd.read_excel("path_to_file.xls", "Sheet1", converters={"MyBools": bool})
```

This options handles missing values and treats exceptions in the converters as missing data. Transformations are applied cell by cell rather than to the column as a whole, so the array dtype is not guaranteed. For instance, a column of integers with missing values cannot be transformed to an array with integer dtype, because NaN is strictly a float. You can manually mask missing data to recover integer dtype:

```
def cfun(x):
    return int(x) if x else -1


pd.read_excel("path_to_file.xls", "Sheet1", converters={"MyInts": cfun})
```

Dtype specifications

As an alternative to converters, the type for an entire column can be specified using the `dtype` keyword, which takes a dictionary mapping column names to types. To interpret data with no type inference, use the type `str` or `object`.

```
pd.read_excel("path_to_file.xls", dtype={"MyInts": "int64", "MyText": str})
```

Writing Excel files

Writing Excel files to disk

To write a `DataFrame` object to a sheet of an Excel file, you can use the `to_excel` instance method. The arguments are largely the same as `to_csv` described above, the first argument being the name of the excel file, and the optional second argument the name of the sheet to which the `DataFrame` should be written. For example:

```
df.to_excel("path_to_file.xlsx", sheet_name="Sheet1")
```

Skip to main content

The `DataFrame` will be written in a way that tries to mimic the REPL output. The `index_label` will be placed in the second row instead of the first. You can place it in the first row by setting the `merge_cells` option in `to_excel()` to `False`:

```
df.to_excel("path_to_file.xlsx", index_label="label", merge_cells=False)
```

In order to write separate `DataFrames` to separate sheets in a single Excel file, one can pass an `ExcelWriter`.

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

When using the `engine_kwargs` parameter, pandas will pass these arguments to the engine. For this, it is important to know which function pandas is using internally.

- For the engine openpyxl, pandas is using `openpyxl.Workbook()` to create a new sheet and `openpyxl.load_workbook()` to append data to an existing sheet. The openpyxl engine writes to (`.xlsx`) and (`.xlsm`) files.
- For the engine xlsxwriter, pandas is using `xlsxwriter.Workbook()` to write to (`.xlsx`) files.
- For the engine odf, pandas is using `odf.opendocument.OpenDocumentSpreadsheet()` to write to (`.ods`) files.

Writing Excel files to memory

pandas supports writing Excel files to buffer-like objects such as `StringIO` or `BytesIO` using `ExcelWriter`.

```
from io import BytesIO

bio = BytesIO()

# By setting the 'engine' in the ExcelWriter constructor.
writer = pd.ExcelWriter(bio, engine="xlsxwriter")
df.to_excel(writer, sheet_name="Sheet1")

# Save the workbook
writer.save()

# Seek to the beginning and read to copy the workbook to a variable in memory
bio.seek(0)
workbook = bio.read()
```

Skip to main content

Note

`engine` is optional but recommended. Setting the engine determines the version of workbook produced. Setting `engine='xlrd'` will produce an Excel 2003-format workbook (xls). Using either `'openpyxl'` or `'xlsxwriter'` will produce an Excel 2007-format workbook (xlsx). If omitted, an Excel 2007-formatted workbook is produced.

Excel writer engines

pandas chooses an Excel writer via two methods:

1. the `engine` keyword argument
2. the filename extension (via the default specified in config options)

By default, pandas uses the XlsxWriter for `.xlsx`, openpyxl for `.xlsm`. If you have multiple engines installed, you can set the default engine through setting the config options `io.excel.xlsx.writer` and `io.excel.xls.writer`. pandas will fall back on openpyxl for `.xlsx` files if Xlsxwriter is not available.

To specify which writer you want to use, you can pass an engine keyword argument to `to_excel` and to `ExcelWriter`. The built-in engines are:

- `openpyxl`: version 2.4 or higher is required
- `xlsxwriter`

```
# By setting the 'engine' in the DataFrame 'to_excel()' methods.
df.to_excel("path_to_file.xlsx", sheet_name="Sheet1", engine="xlsxwriter")

# By setting the 'engine' in the ExcelWriter constructor.
writer = pd.ExcelWriter("path_to_file.xlsx", engine="xlsxwriter")

# Or via pandas configuration.
from pandas import options  # noqa: E402

options.io.excel.xlsx.writer = "xlsxwriter"

df.to_excel("path_to_file.xlsx", sheet_name="Sheet1")
```

Style and formatting

The look and feel of Excel worksheets created from pandas can be modified using the following

Skip to main content

- `float_format` : Format string for floating point numbers (default `None`).
- `freeze_panes` : A tuple of two integers representing the bottommost row and rightmost column to freeze. Each of these parameters is one-based, so (1, 1) will freeze the first row and first column (default `None`).

Using the Xlsxwriter engine provides many options for controlling the format of an Excel worksheet created with the `to_excel` method. Excellent examples can be found in the Xlsxwriter documentation here: https://xlsxwriter.readthedocs.io/working_with_pandas.html

OpenDocument Spreadsheets

The io methods for Excel files also support reading and writing OpenDocument spreadsheets using the odfpy module. The semantics and features for reading and writing OpenDocument spreadsheets match what can be done for Excel files using `engine='odf'`. The optional dependency 'odfpy' needs to be installed.

The `read_excel()` method can read OpenDocument spreadsheets

```
# Returns a DataFrame
pd.read_excel("path_to_file.ods", engine="odf")
```

Similarly, the `to_excel()` method can write OpenDocument spreadsheets

```
# Writes DataFrame to a .ods file
df.to_excel("path_to_file.ods", engine="odf")
```

Binary Excel (.xlsb) files

The `read_excel()` method can also read binary Excel files using the `pyxlsb` module. The semantics and features for reading binary Excel files mostly match what can be done for Excel files using `engine='pyxlsb'`. `pyxlsb` does not recognize datetime types in files and will return floats instead (you can use calamine if you need recognize datetime types).

```
# Returns a DataFrame
pd.read_excel("path_to_file.xlsb", engine="pyxlsb")
```

Skip to main content

Note

Currently pandas only supports *reading* binary Excel files. Writing is not implemented.

Calamine (Excel and ODS files)

The `read_excel()` method can read Excel file (`.xlsx`, `.xlsm`, `.xls`, `.xlsb`) and OpenDocument spreadsheets (`.ods`) using the `python-calamine` module. This module is a binding for Rust library calamine and is faster than other engines in most cases. The optional dependency 'python-calamine' needs to be installed.

```
# Returns a DataFrame
pd.read_excel("path_to_file.xlsb", engine="calamine")
```

Clipboard

A handy way to grab data is to use the `read_clipboard()` method, which takes the contents of the clipboard buffer and passes them to the `read_csv` method. For instance, you can copy the following text to the clipboard (CTRL-C on many operating systems):

```
  A B C
x 1 4 p
y 2 5 q
z 3 6 r
```

And then import the data directly to a `DataFrame` by calling:

```
>>> clipdf = pd.read_clipboard()
>>> clipdf
  A B C
x 1 4 p
y 2 5 q
z 3 6 r
```

The `to_clipboard` method can be used to write the contents of a `DataFrame` to the clipboard. Following which you can paste the clipboard contents into other applications (CTRL-V on many operating systems). Here we illustrate writing a `DataFrame` into clipboard and reading it back.

Skip to main content

```
... )

>>> df
  A B C
x 1 4 p
y 2 5 q
z 3 6 r
>>> df.to_clipboard()
>>> pd.read_clipboard()
  A B C
x 1 4 p
y 2 5 q
z 3 6 r
```

We can see that we got the same content back, which we had earlier written to the clipboard.

Note

You may need to install xclip or xsel (with PyQt5, PyQt4 or qtpy) on Linux to use these methods.

Pickling

All pandas objects are equipped with `to_pickle` methods which use Python's `cPickle` module to save data structures to disk using the pickle format.

```
In [434]: df
Out[434]:
c1         a
c2         b  d
lvl1 lvl2
a    c     1  5
     d     2  6
b    c     3  7
     d     4  8

In [435]: df.to_pickle("foo.pkl")
```

The `read_pickle` function in the `pandas` namespace can be used to load any pickled pandas object (or any other pickled object) from file:

```
In [436]: pd.read_pickle("foo.pkl")
Out[436]:
c1         a
c2         b  d
```

Skip to main content

```
      d     2  6
b     c     3  7
      d     4  8
```

Warning

Loading pickled data received from untrusted sources can be unsafe.

See: https://docs.python.org/3/library/pickle.html

Warning

`read_pickle()` is only guaranteed backwards compatible back to a few minor release.

Compressed pickle files

`read_pickle()`, `DataFrame.to_pickle()` and `Series.to_pickle()` can read and write compressed pickle files. The compression types of `gzip`, `bz2`, `xz`, `zstd` are supported for reading and writing. The `zip` file format only supports reading and must contain only one data file to be read.

The compression type can be an explicit parameter or be inferred from the file extension. If 'infer', then use `gzip`, `bz2`, `zip`, `xz`, `zstd` if filename ends in `'.gz'`, `'.bz2'`, `'.zip'`, `'.xz'`, or `'.zst'`, respectively.

The compression parameter can also be a `dict` in order to pass options to the compression protocol. It must have a `'method'` key set to the name of the compression protocol, which must be one of {`'zip'`, `'gzip'`, `'bz2'`, `'xz'`, `'zstd'`}. All other key-value pairs are passed to the underlying compression library.

```
In [437]: df = pd.DataFrame(
     .....:     {
     .....:         "A": np.random.randn(1000),
     .....:         "B": "foo",
     .....:         "C": pd.date_range("20130101", periods=1000, freq="s"),
     .....:     }
     .....: )
     .....:

In [438]: df
Out[438]:
            A    B                   C
```

Skip to main content

```
2    0.896171  foo 2013-01-01 00:00:02
3   -0.487602  foo 2013-01-01 00:00:03
4   -0.082240  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.171092  foo 2013-01-01 00:16:35
996  1.786173  foo 2013-01-01 00:16:36
997 -0.575189  foo 2013-01-01 00:16:37
998  0.820750  foo 2013-01-01 00:16:38
999 -1.256530  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Using an explicit compression type:

```
In [439]: df.to_pickle("data.pkl.compress", compression="gzip")

In [440]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [441]: rt
Out[441]:
            A    B                   C
0   -0.317441  foo 2013-01-01 00:00:00
1   -1.236269  foo 2013-01-01 00:00:01
2    0.896171  foo 2013-01-01 00:00:02
3   -0.487602  foo 2013-01-01 00:00:03
4   -0.082240  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.171092  foo 2013-01-01 00:16:35
996  1.786173  foo 2013-01-01 00:16:36
997 -0.575189  foo 2013-01-01 00:16:37
998  0.820750  foo 2013-01-01 00:16:38
999 -1.256530  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Inferring compression type from the extension:

```
In [442]: df.to_pickle("data.pkl.xz", compression="infer")

In [443]: rt = pd.read_pickle("data.pkl.xz", compression="infer")

In [444]: rt
Out[444]:
            A    B                   C
0   -0.317441  foo 2013-01-01 00:00:00
1   -1.236269  foo 2013-01-01 00:00:01
2    0.896171  foo 2013-01-01 00:00:02
3   -0.487602  foo 2013-01-01 00:00:03
4   -0.082240  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.171092  foo 2013-01-01 00:16:35
996  1.786173  foo 2013-01-01 00:16:36
997 -0.575189  foo 2013-01-01 00:16:37
```

Skip to main content

```
[1000 rows x 3 columns]
```

The default is to ‘infer’:

```
In [445]: df.to_pickle("data.pkl.gz")

In [446]: rt = pd.read_pickle("data.pkl.gz")

In [447]: rt
Out[447]:
            A    B                   C
0   -0.317441  foo 2013-01-01 00:00:00
1   -1.236269  foo 2013-01-01 00:00:01
2    0.896171  foo 2013-01-01 00:00:02
3   -0.487602  foo 2013-01-01 00:00:03
4   -0.082240  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.171092  foo 2013-01-01 00:16:35
996  1.786173  foo 2013-01-01 00:16:36
997 -0.575189  foo 2013-01-01 00:16:37
998  0.820750  foo 2013-01-01 00:16:38
999 -1.256530  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]

In [448]: df["A"].to_pickle("s1.pkl.bz2")

In [449]: rt = pd.read_pickle("s1.pkl.bz2")

In [450]: rt
Out[450]:
0     -0.317441
1     -1.236269
2      0.896171
3     -0.487602
4     -0.082240
         ...
995   -0.171092
996    1.786173
997   -0.575189
998    0.820750
999   -1.256530
Name: A, Length: 1000, dtype: float64
```

Passing options to the compression protocol in order to speed up compression:

```
In [451]: df.to_pickle("data.pkl.gz", compression={"method": "gzip", "compresslevel": 1
```

Skip to main content

pandas support for `msgpack` has been removed in version 1.0.0. It is recommended to use pickle instead.

Alternatively, you can also the Arrow IPC serialization format for on-the-wire transmission of pandas objects. For documentation on pyarrow, see here.

HDF5 (PyTables)

`HDFStore` is a dict-like object which reads and writes pandas using the high performance HDF5 format using the excellent PyTables library. See the cookbook for some advanced strategies

> **Warning**
>
> pandas uses PyTables for reading and writing HDF5 files, which allows serializing object-dtype data with pickle. Loading pickled data received from untrusted sources can be unsafe.
>
> See: https://docs.python.org/3/library/pickle.html for more.

```
In [452]: store = pd.HDFStore("store.h5")

In [453]: print(store)
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Objects can be written to the file just like adding key-value pairs to a dict:

```
In [454]: index = pd.date_range("1/1/2000", periods=8)

In [455]: s = pd.Series(np.random.randn(5), index=["a", "b", "c", "d", "e"])

In [456]: df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=["A", "B", "C"]

# store.put('s', s) is an equivalent method
In [457]: store["s"] = s

In [458]: store["df"] = df

In [459]: store
Out[459]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Skip to main content

```
# store.get('df') is an equivalent method
In [460]: store["df"]
Out[460]:
                   A         B         C
2000-01-01  0.858644 -0.851236  1.058006
2000-01-02 -0.080372 -1.268121  1.561967
2000-01-03  0.816983  1.965656 -1.169408
2000-01-04  0.712795 -0.062433  0.736755
2000-01-05 -0.298721 -1.988045  1.475308
2000-01-06  1.103675  1.382242 -0.650762
2000-01-07 -0.729161 -0.142928 -1.063038
2000-01-08 -1.005977  0.465222 -0.094517

# dotted (attribute) access provides get as well
In [461]: store.df
Out[461]:
                   A         B         C
2000-01-01  0.858644 -0.851236  1.058006
2000-01-02 -0.080372 -1.268121  1.561967
2000-01-03  0.816983  1.965656 -1.169408
2000-01-04  0.712795 -0.062433  0.736755
2000-01-05 -0.298721 -1.988045  1.475308
2000-01-06  1.103675  1.382242 -0.650762
2000-01-07 -0.729161 -0.142928 -1.063038
2000-01-08 -1.005977  0.465222 -0.094517
```

Deletion of the object specified by the key:

```
# store.remove('df') is an equivalent method
In [462]: del store["df"]

In [463]: store
Out[463]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Closing a Store and using a context manager:

```
In [464]: store.close()

In [465]: store
Out[465]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

In [466]: store.is_open
Out[466]: False

# Working with, and automatically closing the store using a context manager
In [467]: with pd.HDFStore("store.h5") as store:
   .....:     store.keys()
   .....:
```

Skip to main content

Read/write API

`HDFStore` supports a top-level API using `read_hdf` for reading and `to_hdf` for writing, similar to how `read_csv` and `to_csv` work.

```
In [468]: df_tl = pd.DataFrame({"A": list(range(5)), "B": list(range(5))})

In [469]: df_tl.to_hdf("store_tl.h5", key="table", append=True)

In [470]: pd.read_hdf("store_tl.h5", "table", where=["index>2"])
Out[470]:
   A  B
3  3  3
4  4  4
```

HDFStore will by default not drop rows that are all missing. This behavior can be changed by setting `dropna=True`.

```
In [471]: df_with_missing = pd.DataFrame(
   .....:     {
   .....:         "col1": [0, np.nan, 2],
   .....:         "col2": [1, np.nan, np.nan],
   .....:     }
   .....: )
   .....:

In [472]: df_with_missing
Out[472]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [473]: df_with_missing.to_hdf("file.h5", key="df_with_missing", format="table", mode

In [474]: pd.read_hdf("file.h5", "df_with_missing")
Out[474]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [475]: df_with_missing.to_hdf(
   .....:     "file.h5", key="df_with_missing", format="table", mode="w", dropna=True
   .....: )
   .....:

In [476]: pd.read_hdf("file.h5", "df_with_missing")
Out[476]:
   col1  col2
0   0.0   1.0
2   2.0   NaN
```

Skip to main content

Fixed format

The examples above show storing using `put`, which write the HDF5 to `PyTables` in a fixed array format, called the `fixed` format. These types of stores are **not** appendable once written (though you can simply remove them and rewrite). Nor are they **queryable**; they must be retrieved in their entirety. They also do not support dataframes with non-unique column names. The `fixed` format stores offer very fast writing and slightly faster reading than `table` stores. This format is specified by default when using `put` or `to_hdf` or by `format='fixed'` or `format='f'`.

Skip to main content

Warning

A `fixed` format will raise a `TypeError` if you try to retrieve using a `where`:

```
In [477]: pd.DataFrame(np.random.randn(10, 2)).to_hdf("test_fixed.h5", key="df

In [478]: pd.read_hdf("test_fixed.h5", "df", where="index>5")
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[478], line 1
----> 1 pd.read_hdf("test_fixed.h5", "df", where="index>5")

File /home/pandas/pandas/io/pytables.py:452, in read_hdf(path_or_buf, key, mod
    447                 raise ValueError(
    448                     "key must be provided when HDF5 "
    449                     "file contains multiple datasets."
    450                 )
    451             key = candidate_only_group._v_pathname
--> 452         return store.select(
    453             key,
    454             where=where,
    455             start=start,
    456             stop=stop,
    457             columns=columns,
    458             iterator=iterator,
    459             chunksize=chunksize,
    460             auto_close=auto_close,
    461         )
    462     except (ValueError, TypeError, LookupError):
    463         if not isinstance(path_or_buf, HDFStore):
    464             # if there is an error, close the store if we opened it.

File /home/pandas/pandas/io/pytables.py:906, in HDFStore.select(self, key, whe
    892 # create the iterator
    893 it = TableIterator(
    894     self,
    895     s,
   (...)
    903     auto_close=auto_close,
    904 )
--> 906 return it.get_result()

File /home/pandas/pandas/io/pytables.py:2029, in TableIterator.get_result(self
   2026     where = self.where
   2028 # directly return the result
-> 2029 results = self.func(self.start, self.stop, where)
   2030 self.close()
   2031 return results

File /home/pandas/pandas/io/pytables.py:890, in HDFStore.select.<locals>.func(
    889 def func(_start, _stop, _where):
--> 890     return s.read(start=_start, stop=_stop, where=_where, columns=colu

File /home/pandas/pandas/io/pytables.py:3278, in BlockManagerFixed.read(self, 
   3270 def read(
```

Skip to main content

```
    (...)
   3276 ) -> DataFrame:
   3277     # start, stop applied to rows, so 0th axis only
-> 3278     self.validate_read(columns, where)
   3279     select_axis = self.obj_type()._get_block_manager_axis(0)
   3281     axes = []

File /home/pandas/pandas/io/pytables.py:2920, in GenericFixed.validate_read(se
   2915     raise TypeError(
   2916         "cannot pass a column specification when reading "
   2917         "a Fixed format store. this store must be selected in its entir
   2918     )
   2919 if where is not None:
-> 2920     raise TypeError(
   2921         "cannot pass a where specification when reading "
   2922         "from a Fixed format store. this store must be selected in its
   2923     )

TypeError: cannot pass a where specification when reading from a Fixed format s
```

Table format

`HDFStore` supports another `PyTables` format on disk, the `table` format. Conceptually a `table` is shaped very much like a DataFrame, with rows and columns. A `table` may be appended to in the same or other sessions. In addition, delete and query type operations are supported. This format is specified by `format='table'` or `format='t'` to `append` or `put` or `to_hdf`.

This format can be set as an option as well `pd.set_option('io.hdf.default_format','table')` to enable `put/append/to_hdf` to by default store in the `table` format.

```
In [479]: store = pd.HDFStore("store.h5")

In [480]: df1 = df[0:4]

In [481]: df2 = df[4:]

# append data (creates a table automatically)
In [482]: store.append("df", df1)

In [483]: store.append("df", df2)

In [484]: store
Out[484]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# select the entire object
In [485]: store.select("df")
```

Skip to main content

```
2000-01-01  0.858644 -0.851236  1.058006
2000-01-02 -0.080372 -1.268121  1.561967
2000-01-03  0.816983  1.965656 -1.169408
2000-01-04  0.712795 -0.062433  0.736755
2000-01-05 -0.298721 -1.988045  1.475308
2000-01-06  1.103675  1.382242 -0.650762
2000-01-07 -0.729161 -0.142928 -1.063038
2000-01-08 -1.005977  0.465222 -0.094517

# the type of stored data
In [486]: store.root.df._v_attrs.pandas_type
Out[486]: 'frame_table'
```

Note

You can also create a `table` by passing `format='table'` or `format='t'` to a `put` operation.

Hierarchical keys

Keys to a store can be specified as a string. These can be in a hierarchical path-name like format (e.g. `foo/bar/bah`), which will generate a hierarchy of sub-stores (or `Groups` in PyTables parlance). Keys can be specified without the leading '/' and are **always** absolute (e.g. 'foo' refers to '/foo'). Removal operations can remove everything in the sub-store and **below**, so be *careful*.

```
In [487]: store.put("foo/bar/bah", df)

In [488]: store.append("food/orange", df)

In [489]: store.append("food/apple", df)

In [490]: store
Out[490]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# a list of keys are returned
In [491]: store.keys()
Out[491]: ['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [492]: store.remove("food")

In [493]: store
Out[493]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Skip to main content

You can walk through the group hierarchy using the `walk` method which will yield a tuple for each group key along with the relative keys of its contents.

```
In [494]: for (path, subgroups, subkeys) in store.walk():
   .....:     for subgroup in subgroups:
   .....:         print("GROUP: {}/{}".format(path, subgroup))
   .....:     for subkey in subkeys:
   .....:         key = "/".join([path, subkey])
   .....:         print("KEY: {}".format(key))
   .....:         print(store.get(key))
   .....:
GROUP: /foo
KEY: /df
                   A         B         C
2000-01-01  0.858644 -0.851236  1.058006
2000-01-02 -0.080372 -1.268121  1.561967
2000-01-03  0.816983  1.965656 -1.169408
2000-01-04  0.712795 -0.062433  0.736755
2000-01-05 -0.298721 -1.988045  1.475308
2000-01-06  1.103675  1.382242 -0.650762
2000-01-07 -0.729161 -0.142928 -1.063038
2000-01-08 -1.005977  0.465222 -0.094517
GROUP: /foo/bar
KEY: /foo/bar/bah
                   A         B         C
2000-01-01  0.858644 -0.851236  1.058006
2000-01-02 -0.080372 -1.268121  1.561967
2000-01-03  0.816983  1.965656 -1.169408
2000-01-04  0.712795 -0.062433  0.736755
2000-01-05 -0.298721 -1.988045  1.475308
2000-01-06  1.103675  1.382242 -0.650762
2000-01-07 -0.729161 -0.142928 -1.063038
2000-01-08 -1.005977  0.465222 -0.094517
```

Skip to main content

Warning

Hierarchical keys cannot be retrieved as dotted (attribute) access as described above for items stored under the root node.

```
In [495]: store.foo.bar.bah
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[495], line 1
----> 1 store.foo.bar.bah

File /home/pandas/pandas/io/pytables.py:613, in HDFStore.__getattr__(self, nam
    611 """allow attribute access to get stores"""
    612 try:
--> 613     return self.get(name)
    614 except (KeyError, ClosedFileError):
    615     pass

File /home/pandas/pandas/io/pytables.py:813, in HDFStore.get(self, key)
    811 if group is None:
    812     raise KeyError(f"No object named {key} in the file")
--> 813 return self._read_group(group)

File /home/pandas/pandas/io/pytables.py:1878, in HDFStore._read_group(self, gr
   1877 def _read_group(self, group: Node):
-> 1878     s = self._create_storer(group)
   1879     s.infer_axes()
   1880     return s.read()

File /home/pandas/pandas/io/pytables.py:1752, in HDFStore._create_storer(self,
   1750         tt = "generic_table"
   1751     else:
-> 1752         raise TypeError(
   1753             "cannot create a storer if the object is not existing "
   1754             "nor a value are passed"
   1755         )
   1756 else:
   1757     if isinstance(value, Series):

TypeError: cannot create a storer if the object is not existing nor a value ar
```

```
# you can directly access the actual PyTables node but using the root node
In [496]: store.root.foo.bar.bah
Out[496]:
/foo/bar/bah (Group) ''
  children := ['axis0' (Array), 'axis1' (Array), 'block0_items' (Array), 'bloc
```

Instead, use explicit string based keys:

Skip to main content

```
                   A         B         C
2000-01-01  0.858644 -0.851236  1.058006
2000-01-02 -0.080372 -1.268121  1.561967
2000-01-03  0.816983  1.965656 -1.169408
2000-01-04  0.712795 -0.062433  0.736755
2000-01-05 -0.298721 -1.988045  1.475308
2000-01-06  1.103675  1.382242 -0.650762
2000-01-07 -0.729161 -0.142928 -1.063038
2000-01-08 -1.005977  0.465222 -0.094517
```

Storing types

Storing mixed types in a table

Storing mixed-dtype data is supported. Strings are stored as a fixed-width using the maximum size of the appended column. Subsequent attempts at appending longer strings will raise a `ValueError`.

Passing `min_itemsize={`values`: size}` as a parameter to append will set a larger minimum for the string columns. Storing `floats, strings, ints, bools, datetime64` are currently supported. For string columns, passing `nan_rep = 'nan'` to append will change the default nan representation on disk (which converts to/from `np.nan`), this defaults to `nan`.

```
In [498]: df_mixed = pd.DataFrame(
   .....:     {
   .....:         "A": np.random.randn(8),
   .....:         "B": np.random.randn(8),
   .....:         "C": np.array(np.random.randn(8), dtype="float32"),
   .....:         "string": "string",
   .....:         "int": 1,
   .....:         "bool": True,
   .....:         "datetime64": pd.Timestamp("20010102"),
   .....:     },
   .....:     index=list(range(8)),
   .....: )
   .....:

In [499]: df_mixed.loc[df_mixed.index[3:5], ["A", "B", "string", "datetime64"]] = np.na

In [500]: store.append("df_mixed", df_mixed, min_itemsize={"values": 50})

In [501]: df_mixed1 = store.select("df_mixed")

In [502]: df_mixed1
Out[502]:
          A         B         C  string  int  bool                    datetime64
0  0.013747 -1.166078 -1.292080  string    1  True 1970-01-01 00:00:00.978393600
```

Skip to main content

```
3       NaN       NaN  0.097771     NaN    1  True                          NaT
4       NaN       NaN  1.536408     NaN    1  True                          NaT
5 -0.023202  0.043702  0.926790  string    1  True 1970-01-01 00:00:00.978393600
6  2.359782  0.088224 -0.676448  string    1  True 1970-01-01 00:00:00.978393600
7 -0.143428 -0.813360 -0.179724  string    1  True 1970-01-01 00:00:00.978393600

In [503]: df_mixed1.dtypes.value_counts()
Out[503]:
float64           2
float32           1
object            1
int64             1
bool              1
datetime64[ns]    1
Name: count, dtype: int64

# we have provided a minimum string column size
In [504]: store.root.df_mixed.table
Out[504]:
/df_mixed/table (Table(8,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(2,), dflt=0.0, pos=1),
  "values_block_1": Float32Col(shape=(1,), dflt=0.0, pos=2),
  "values_block_2": StringCol(itemsize=50, shape=(1,), dflt=b'', pos=3),
  "values_block_3": Int64Col(shape=(1,), dflt=0, pos=4),
  "values_block_4": BoolCol(shape=(1,), dflt=False, pos=5),
  "values_block_5": Int64Col(shape=(1,), dflt=0, pos=6)}
  byteorder := 'little'
  chunkshape := (689,)
  autoindex := True
  colindexes := {
    "index": Index(6, mediumshuffle, zlib(1)).is_csi=False}
```

Storing MultiIndex DataFrames

Storing MultiIndex `DataFrames` as tables is very similar to storing/selecting from homogeneous index `DataFrames`.

```
In [505]: index = pd.MultiIndex(
   .....:     levels=[["foo", "bar", "baz", "qux"], ["one", "two", "three"]],
   .....:     codes=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3], [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   .....:     names=["foo", "bar"],
   .....: )
   .....:

In [506]: df_mi = pd.DataFrame(np.random.randn(10, 3), index=index, columns=["A", "B",

In [507]: df_mi
Out[507]:
                A         B         C
foo bar
```

Skip to main content

```
    two    1.012694  0.414147  1.950460
    three  1.094544 -0.802899 -0.583343
bar one    0.410395  0.618321  0.560398
    two    1.434027 -0.033270  0.343197
baz two   -1.646063 -0.695847 -0.429156
    three -0.244688 -1.428229 -0.138691
qux one    1.866184 -1.446617  0.036660
    two   -1.660522  0.929553 -1.298649
    three  3.565769  0.682402  1.041927

In [508]: store.append("df_mi", df_mi)

In [509]: store.select("df_mi")
Out[509]:
                  A         B         C
foo bar
foo one   -1.303456 -0.642994 -0.649456
    two    1.012694  0.414147  1.950460
    three  1.094544 -0.802899 -0.583343
bar one    0.410395  0.618321  0.560398
    two    1.434027 -0.033270  0.343197
baz two   -1.646063 -0.695847 -0.429156
    three -0.244688 -1.428229 -0.138691
qux one    1.866184 -1.446617  0.036660
    two   -1.660522  0.929553 -1.298649
    three  3.565769  0.682402  1.041927

# the levels are automatically included as data columns
In [510]: store.select("df_mi", "foo=bar")
Out[510]:
                A         B         C
foo bar
bar one  0.410395  0.618321  0.560398
    two  1.434027 -0.033270  0.343197
```

Note

The `index` keyword is reserved and cannot be use as a level name.

Querying

Querying a table

`select` and `delete` operations have an optional criterion that can be specified to select/delete only a subset of the data. This allows one to have a very large on-disk table and retrieve only a portion of the data.

Skip to main content

- `index` and `columns` are supported indexers of `DataFrames`.
- if `data_columns` are specified, these can be used as additional indexers.
- level name in a MultiIndex, with default name `level_0`, `level_1`, … if not provided.

Valid comparison operators are:

`=, ==, !=, >, >=, <, <=`

Valid boolean expressions are combined with:

- `|` : or
- `&` : and
- `(` and `)` : for grouping

These rules are similar to how boolean expressions are used in pandas for indexing.

Note

- `=` will be automatically expanded to the comparison operator `==`
- `~` is the not operator, but can only be used in very limited circumstances
- If a list/tuple of expressions is passed they will be combined via `&`

The following are valid expressions:

- `'index >= date'`
- `"columns = ['A', 'D']"`
- `"columns in ['A', 'D']"`
- `'columns = A'`
- `'columns == A'`
- `"~(columns = ['A', 'B'])"`
- `'index > df.index[3] & string = "bar"'`
- `'(index > df.index[3] & index <= df.index[6]) | string = "bar"'`
- `"ts >= Timestamp('2012-02-01')"`
- `"major_axis>=20130101"`

The `indexers` are on the left-hand side of the sub-expression:

`columns`, `major_axis`, `ts`

Skip to main content

- functions that will be evaluated, e.g. `Timestamp('2012-02-01')`
- strings, e.g. `"bar"`
- date-like, e.g. `20130101`, or `"20130101"`
- lists, e.g. `"['A', 'B']"`
- variables that are defined in the local names space, e.g. `date`

Note

Passing a string to a query by interpolating it into the query expression is not recommended. Simply assign the string of interest to a variable and use that variable in an expression. For example, do this

```
string = "HolyMoly'"
store.select("df", "index == string")
```

instead of this

```
string = "HolyMoly'"
store.select('df', f'index == {string}')
```

The latter will **not** work and will raise a `SyntaxError`.Note that there's a single quote followed by a double quote in the `string` variable.

If you *must* interpolate, use the `'%r'` format specifier

```
store.select("df", "index == %r" % string)
```

which will quote `string`.

Here are some examples:

```
In [511]: dfq = pd.DataFrame(
   .....:     np.random.randn(10, 4),
   .....:     columns=list("ABCD"),
   .....:     index=pd.date_range("20130101", periods=10),
   .....: )
   .....:

In [512]: store.append("dfq", dfq, format="table", data_columns=True)
```

Skip to main content

```
In [513]: store.select("dfq", "index>pd.Timestamp('20130104') & columns=['A', 'B']")
Out[513]:
                   A         B
2013-01-05 -0.830545 -0.457071
2013-01-06  0.431186  1.049421
2013-01-07  0.617509 -0.811230
2013-01-08  0.947422 -0.671233
2013-01-09 -0.183798 -1.211230
2013-01-10  0.361428  0.887304
```

Use inline column reference.

```
In [514]: store.select("dfq", where="A>0 or C>0")
Out[514]:
                   A         B         C         D
2013-01-02  0.658179  0.362814 -0.917897  0.010165
2013-01-03  0.905122  1.848731 -1.184241  0.932053
2013-01-05 -0.830545 -0.457071  1.565581  1.148032
2013-01-06  0.431186  1.049421  0.383309  0.595013
2013-01-07  0.617509 -0.811230 -2.088563 -1.393500
2013-01-08  0.947422 -0.671233 -0.847097 -1.187785
2013-01-10  0.361428  0.887304  0.266457 -0.399641
```

The `columns` keyword can be supplied to select a list of columns to be returned, this is equivalent to passing a `'columns=list_of_columns_to_filter'`:

```
In [515]: store.select("df", "columns=['A', 'B']")
Out[515]:
                   A         B
2000-01-01  0.858644 -0.851236
2000-01-02 -0.080372 -1.268121
2000-01-03  0.816983  1.965656
2000-01-04  0.712795 -0.062433
2000-01-05 -0.298721 -1.988045
2000-01-06  1.103675  1.382242
2000-01-07 -0.729161 -0.142928
2000-01-08 -1.005977  0.465222
```

`start` and `stop` parameters can be specified to limit the total search space. These are in terms of the total number of rows in a table.

Skip to main content

Note

`select` will raise a `ValueError` if the query expression has an unknown variable reference. Usually this means that you are trying to select on a column that is **not** a data_column.

`select` will raise a `SyntaxError` if the query expression is not valid.

Query timedelta64[ns]

You can store and query using the `timedelta64[ns]` type. Terms can be specified in the format: `<float>(<unit>)`, where float may be signed (and fractional), and unit can be `D,s,ms,us,ns` for the timedelta. Here's an example:

```
In [516]: from datetime import timedelta

In [517]: dftd = pd.DataFrame(
   .....:     {
   .....:         "A": pd.Timestamp("20130101"),
   .....:         "B": [
   .....:             pd.Timestamp("20130101") + timedelta(days=i, seconds=10)
   .....:             for i in range(10)
   .....:         ],
   .....:     }
   .....: )
   .....:

In [518]: dftd["C"] = dftd["A"] - dftd["B"]

In [519]: dftd
Out[519]:
           A                   B                   C
0 2013-01-01 2013-01-01 00:00:10  -1 days +23:59:50
1 2013-01-01 2013-01-02 00:00:10  -2 days +23:59:50
2 2013-01-01 2013-01-03 00:00:10  -3 days +23:59:50
3 2013-01-01 2013-01-04 00:00:10  -4 days +23:59:50
4 2013-01-01 2013-01-05 00:00:10  -5 days +23:59:50
5 2013-01-01 2013-01-06 00:00:10  -6 days +23:59:50
6 2013-01-01 2013-01-07 00:00:10  -7 days +23:59:50
7 2013-01-01 2013-01-08 00:00:10  -8 days +23:59:50
8 2013-01-01 2013-01-09 00:00:10  -9 days +23:59:50
9 2013-01-01 2013-01-10 00:00:10 -10 days +23:59:50

In [520]: store.append("dftd", dftd, data_columns=True)

In [521]: store.select("dftd", "C<'-3.5D'")
Out[521]:
                           A                   B                   C
4 1970-01-01 00:00:01.356998400 2013-01-05 00:00:10  -5 days +23:59:50
```

Skip to main content

```
7 1970-01-01 00:00:01.356998400 2013-01-08 00:00:10  -8 days +23:59:50
8 1970-01-01 00:00:01.356998400 2013-01-09 00:00:10  -9 days +23:59:50
9 1970-01-01 00:00:01.356998400 2013-01-10 00:00:10 -10 days +23:59:50
```

Query MultiIndex

Selecting from a `MultiIndex` can be achieved by using the name of the level.

```
In [522]: df_mi.index.names
Out[522]: FrozenList(['foo', 'bar'])

In [523]: store.select("df_mi", "foo=baz and bar=two")
Out[523]:
                A         B         C
foo bar
baz two -1.646063 -0.695847 -0.429156
```

If the `MultiIndex` levels names are `None`, the levels are automatically made available via the `level_n` keyword with `n` the level of the `MultiIndex` you want to select from.

```
In [524]: index = pd.MultiIndex(
   .....:     levels=[["foo", "bar", "baz", "qux"], ["one", "two", "three"]],
   .....:     codes=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3], [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   .....: )
   .....:

In [525]: df_mi_2 = pd.DataFrame(np.random.randn(10, 3), index=index, columns=["A", "B"

In [526]: df_mi_2
Out[526]:
                  A         B         C
foo one   -0.219582  1.186860 -1.437189
    two    0.053768  1.872644 -1.469813
    three -0.564201  0.876341  0.407749
bar one   -0.232583  0.179812  0.922152
    two   -1.820952 -0.641360  2.133239
baz two   -0.941248 -0.136307 -1.271305
    three -0.099774 -0.061438 -0.845172
qux one    0.465793  0.756995 -0.541690
    two   -0.802241  0.877657 -2.553831
    three  0.094899 -2.319519  0.293601

In [527]: store.append("df_mi_2", df_mi_2)

# the levels are automatically included as data columns with keyword level_n
In [528]: store.select("df_mi_2", "level_0=foo and level_1=two")
Out[528]:
                A         B         C
foo two  0.053768  1.872644 -1.469813
```

Skip to main content

Indexing

You can create/modify an index for a table with `create_table_index` after data is already in the table (after and `append/put` operation). Creating a table index is **highly** encouraged. This will speed your queries a great deal when you use a `select` with the indexed dimension as the `where`.

> **Note**
>
> Indexes are automagically created on the indexables and any data columns you specify. This behavior can be turned off by passing `index=False` to `append`.

```
# we have automagically already created an index (in the first section)
In [529]: i = store.root.df.table.cols.index.index

In [530]: i.optlevel, i.kind
Out[530]: (6, 'medium')

# change an index by passing new parameters
In [531]: store.create_table_index("df", optlevel=9, kind="full")

In [532]: i = store.root.df.table.cols.index.index

In [533]: i.optlevel, i.kind
Out[533]: (9, 'full')
```

Oftentimes when appending large amounts of data to a store, it is useful to turn off index creation for each append, then recreate at the end.

```
In [534]: df_1 = pd.DataFrame(np.random.randn(10, 2), columns=list("AB"))

In [535]: df_2 = pd.DataFrame(np.random.randn(10, 2), columns=list("AB"))

In [536]: st = pd.HDFStore("appends.h5", mode="w")

In [537]: st.append("df", df_1, data_columns=["B"], index=False)

In [538]: st.append("df", df_2, data_columns=["B"], index=False)

In [539]: st.get_storer("df").table
Out[539]:
/df/table (Table(20,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2)}
  byteorder := 'little'
  chunkshape := (2730,)
```

Skip to main content

Then create the index when finished appending.

```
In [540]: st.create_table_index("df", columns=["B"], optlevel=9, kind="full")

In [541]: st.get_storer("df").table
Out[541]:
/df/table (Table(20,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2)}
  byteorder := 'little'
  chunkshape := (2730,)
  autoindex := True
  colindexes := {
    "B": Index(9, fullshuffle, zlib(1)).is_csi=True}

In [542]: st.close()
```

See here for how to create a completely-sorted-index (CSI) on an existing store.

Query via data columns

You can designate (and index) certain columns that you want to be able to perform queries (other than the `indexable` columns, which you can always query). For instance say you want to perform this common operation, on-disk, and return just the frame that matches this query. You can specify `data_columns = True` to force all columns to be `data_columns`.

```
In [543]: df_dc = df.copy()

In [544]: df_dc["string"] = "foo"

In [545]: df_dc.loc[df_dc.index[4:6], "string"] = np.nan

In [546]: df_dc.loc[df_dc.index[7:9], "string"] = "bar"

In [547]: df_dc["string2"] = "cool"

In [548]: df_dc.loc[df_dc.index[1:3], ["B", "C"]] = 1.0

In [549]: df_dc
Out[549]:
                   A         B         C string string2
2000-01-01  0.858644 -0.851236  1.058006    foo    cool
2000-01-02 -0.080372  1.000000  1.000000    foo    cool
2000-01-03  0.816983  1.000000  1.000000    foo    cool
2000-01-04  0.712795 -0.062433  0.736755    foo    cool
2000-01-05 -0.298721 -1.988045  1.475308    NaN    cool
2000-01-06  1.103675  1.382242 -0.650762    NaN    cool
```

Skip to main content

```
# on-disk operations
In [550]: store.append("df_dc", df_dc, data_columns=["B", "C", "string", "string2"])

In [551]: store.select("df_dc", where="B > 0")
Out[551]:
                   A         B         C string string2
2000-01-02 -0.080372  1.000000  1.000000    foo    cool
2000-01-03  0.816983  1.000000  1.000000    foo    cool
2000-01-06  1.103675  1.382242 -0.650762    NaN    cool
2000-01-08 -1.005977  0.465222 -0.094517    bar    cool

# getting creative
In [552]: store.select("df_dc", "B > 0 & C > 0 & string == foo")
Out[552]:
                   A    B    C string string2
2000-01-02 -0.080372  1.0  1.0    foo    cool
2000-01-03  0.816983  1.0  1.0    foo    cool

# this is in-memory version of this type of selection
In [553]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == "foo")]
Out[553]:
                   A    B    C string string2
2000-01-02 -0.080372  1.0  1.0    foo    cool
2000-01-03  0.816983  1.0  1.0    foo    cool

# we have automagically created this index and the B/C/string/string2
# columns are stored separately as ``PyTables`` columns
In [554]: store.root.df_dc.table
Out[554]:
/df_dc/table (Table(8,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2),
  "C": Float64Col(shape=(), dflt=0.0, pos=3),
  "string": StringCol(itemsize=3, shape=(), dflt=b'', pos=4),
  "string2": StringCol(itemsize=4, shape=(), dflt=b'', pos=5)}
  byteorder := 'little'
  chunkshape := (1680,)
  autoindex := True
  colindexes := {
    "index": Index(6, mediumshuffle, zlib(1)).is_csi=False,
    "B": Index(6, mediumshuffle, zlib(1)).is_csi=False,
    "C": Index(6, mediumshuffle, zlib(1)).is_csi=False,
    "string": Index(6, mediumshuffle, zlib(1)).is_csi=False,
    "string2": Index(6, mediumshuffle, zlib(1)).is_csi=False}
```

There is some performance degradation by making lots of columns into `data columns`, so it is up to the user to designate these. In addition, you cannot change data columns (nor indexables) after the first append/put operation (Of course you can simply read in the data and create a new table!).

Skip to main content

You can pass `iterator=True` or `chunksize=number_in_a_chunk` to `select` and `select_as_multiple` to return an iterator on the results. The default is 50,000 rows returned in a chunk.

```
In [555]: for df in store.select("df", chunksize=3):
   .....:     print(df)
   .....:
                   A         B         C
2000-01-01  0.858644 -0.851236  1.058006
2000-01-02 -0.080372 -1.268121  1.561967
2000-01-03  0.816983  1.965656 -1.169408
                   A         B         C
2000-01-04  0.712795 -0.062433  0.736755
2000-01-05 -0.298721 -1.988045  1.475308
2000-01-06  1.103675  1.382242 -0.650762
                   A         B         C
2000-01-07 -0.729161 -0.142928 -1.063038
2000-01-08 -1.005977  0.465222 -0.094517
```

Note

You can also use the iterator with `read_hdf` which will open, then automatically close the store when finished iterating.

```
for df in pd.read_hdf("store.h5", "df", chunksize=3):
    print(df)
```

Note, that the chunksize keyword applies to the **source** rows. So if you are doing a query, then the chunksize will subdivide the total rows in the table and the query applied, returning an iterator on potentially unequal sized chunks.

Here is a recipe for generating a query and using it to create equal sized return chunks.

```
In [556]: dfeq = pd.DataFrame({"number": np.arange(1, 11)})

In [557]: dfeq
Out[557]:
   number
0       1
1       2
2       3
3       4
4       5
5       6
6       7
7       8
```

Skip to main content

```
In [558]: store.append("dfeq", dfeq, data_columns=["number"])

In [559]: def chunks(l, n):
   .....:     return [l[i: i + n] for i in range(0, len(l), n)]
   .....:

In [560]: evens = [2, 4, 6, 8, 10]

In [561]: coordinates = store.select_as_coordinates("dfeq", "number=evens")

In [562]: for c in chunks(coordinates, 2):
   .....:     print(store.select("dfeq", where=c))
   .....:
   number
1       2
3       4
   number
5       6
7       8
   number
9      10
```

Advanced queries

Select a single column

To retrieve a single indexable or data column, use the method `select_column`. This will, for example, enable you to get the index very quickly. These return a `Series` of the result, indexed by the row number. These do not currently accept the `where` selector.

```
In [563]: store.select_column("df_dc", "index")
Out[563]:
0   2000-01-01
1   2000-01-02
2   2000-01-03
3   2000-01-04
4   2000-01-05
5   2000-01-06
6   2000-01-07
7   2000-01-08
Name: index, dtype: datetime64[ns]

In [564]: store.select_column("df_dc", "string")
Out[564]:
0    foo
1    foo
2    foo
3    foo
4    NaN
```

Skip to main content

```
7    bar
Name: string, dtype: object
```

Selecting coordinates

Sometimes you want to get the coordinates (a.k.a the index locations) of your query. This returns an `Index` of the resulting locations. These coordinates can also be passed to subsequent `where` operations.

```
In [565]: df_coord = pd.DataFrame(
   .....:     np.random.randn(1000, 2), index=pd.date_range("20000101", periods=1000)
   .....: )
   .....:

In [566]: store.append("df_coord", df_coord)

In [567]: c = store.select_as_coordinates("df_coord", "index > 20020101")

In [568]: c
Out[568]:
Index([732, 733, 734, 735, 736, 737, 738, 739, 740, 741,
       ...
       990, 991, 992, 993, 994, 995, 996, 997, 998, 999],
      dtype='int64', length=268)

In [569]: store.select("df_coord", where=c)
Out[569]:
                   0         1
2002-01-02  0.007717  1.168386
2002-01-03  0.759328 -0.638934
2002-01-04 -1.154018 -0.324071
2002-01-05 -0.804551 -1.280593
2002-01-06 -0.047208  1.260503
...              ...       ...
2002-09-22 -1.139583  0.344316
2002-09-23 -0.760643 -1.306704
2002-09-24  0.059018  1.775482
2002-09-25  1.242255 -0.055457
2002-09-26  0.410317  2.194489

[268 rows x 2 columns]
```

Selecting using a where mask

Sometime your query can involve creating a list of rows to select. Usually this `mask` would be a resulting `index` from an indexing operation. This example selects the months of a datetimeindex which are 5.

Skip to main content

```
In [570]: df_mask = pd.DataFrame(
   .....:     np.random.randn(1000, 2), index=pd.date_range("20000101", periods=1000)
   .....: )
   .....:

In [571]: store.append("df_mask", df_mask)

In [572]: c = store.select_column("df_mask", "index")

In [573]: where = c[pd.DatetimeIndex(c).month == 5].index

In [574]: store.select("df_mask", where=where)
Out[574]:
                   0         1
2000-05-01  1.479511  0.516433
2000-05-02 -0.334984 -1.493537
2000-05-03  0.900321  0.049695
2000-05-04  0.614266 -1.077151
2000-05-05  0.233881  0.493246
...              ...       ...
2002-05-27  0.294122  0.457407
2002-05-28 -1.102535  1.215650
2002-05-29 -0.432911  0.753606
2002-05-30 -1.105212  2.311877
2002-05-31  2.567296  2.610691

[93 rows x 2 columns]
```

Storer object

If you want to inspect the stored object, retrieve via `get_storer`. You could use this programmatically to say get the number of rows in an object.

```
In [575]: store.get_storer("df_dc").nrows
Out[575]: 8
```

Multiple table queries

The methods `append_to_multiple` and `select_as_multiple` can perform appending/selecting from multiple tables at once. The idea is to have one table (call it the selector table) that you index most/all of the columns, and perform your queries. The other table(s) are data tables with an index matching the selector table's index. You can then perform a very fast query on the selector table, yet get lots of data back. This method is similar to having a very wide table, but enables more efficient queries.

Skip to main content

The `append_to_multiple` method splits a given single DataFrame into multiple tables according to `d`, a dictionary that maps the table names to a list of 'columns' you want in that table. If `None` is used in place of a list, that table will have the remaining unspecified columns of the given DataFrame. The argument `selector` defines which table is the selector table (which you can make queries from). The argument `dropna` will drop rows from the input `DataFrame` to ensure tables are synchronized. This means that if a row for one of the tables being written to is entirely `np.nan`, that row will be dropped from all tables.

If `dropna` is False, **THE USER IS RESPONSIBLE FOR SYNCHRONIZING THE TABLES**. Remember that entirely `np.Nan` rows are not written to the HDFStore, so if you choose to call `dropna=False`, some tables may have more rows than others, and therefore `select_as_multiple` may not work or it may return unexpected results.

```
In [576]: df_mt = pd.DataFrame(
   .....:     np.random.randn(8, 6),
   .....:     index=pd.date_range("1/1/2000", periods=8),
   .....:     columns=["A", "B", "C", "D", "E", "F"],
   .....: )
   .....:

In [577]: df_mt["foo"] = "bar"

In [578]: df_mt.loc[df_mt.index[1], ("A", "B")] = np.nan

# you can also create the tables individually
In [579]: store.append_to_multiple(
   .....:     {"df1_mt": ["A", "B"], "df2_mt": None}, df_mt, selector="df1_mt"
   .....: )
   .....:

In [580]: store
Out[580]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# individual tables were created
In [581]: store.select("df1_mt")
Out[581]:
                   A         B
2000-01-01  0.162291 -0.430489
2000-01-02       NaN       NaN
2000-01-03  0.429207 -1.099274
2000-01-04  1.869081 -1.466039
2000-01-05  0.092130 -1.726280
2000-01-06  0.266901 -0.036854
2000-01-07 -0.517871 -0.990317
2000-01-08 -0.231342  0.557402

In [582]: store.select("df2_mt")
Out[582]:
                   C         D         E         F  foo
```

Skip to main content

```
2000-01-03 -1.069546  1.236277  0.116634 -1.772519  bar
2000-01-04  0.137462  0.313939  0.748471 -0.943009  bar
2000-01-05  0.836517  2.049798  0.562167  0.189952  bar
2000-01-06  1.112750 -0.151596  1.503311  0.939470  bar
2000-01-07 -0.294348  0.335844 -0.794159  1.495614  bar
2000-01-08  0.860312 -0.538674 -0.541986 -1.759606  bar

# as a multiple
In [583]: store.select_as_multiple(
   .....:     ["df1_mt", "df2_mt"],
   .....:     where=["A>0", "B>0"],
   .....:     selector="df1_mt",
   .....: )
   .....:
Out[583]:
Empty DataFrame
Columns: [A, B, C, D, E, F, foo]
Index: []
```

Delete from a table

You can delete from a table selectively by specifying a `where`. In deleting rows, it is important to understand the `PyTables` deletes rows by erasing the rows, then **moving** the following data. Thus deleting can potentially be a very expensive operation depending on the orientation of your data. To get optimal performance, it's worthwhile to have the dimension you are deleting be the first of the `indexables`.

Data is ordered (on the disk) in terms of the `indexables`. Here's a simple use case. You store panel-type data, with dates in the `major_axis` and ids in the `minor_axis`. The data is then interleaved like this:

- **date_1**
    - id_1
    - id_2
    - .
    - id_n
- **date_2**
    - id_1
    - .
    - id_n

It should be clear that a delete operation on the `major_axis` will be fairly quick, as one chunk is

Skip to main content

`minor_axis` will be very expensive. In this case it would almost certainly be faster to rewrite the table using a `where` that selects all but the missing data.

Warning

Please note that HDF5 **DOES NOT RECLAIM SPACE** in the h5 files automatically. Thus, repeatedly deleting (or removing nodes) and adding again, **WILL TEND TO INCREASE THE FILE SIZE**.

To *repack and clean* the file, use ptrepack.

Notes & caveats

Compression

`PyTables` allows the stored data to be compressed. This applies to all kinds of stores, not just tables. Two parameters are used to control compression: `complevel` and `complib`.

- `complevel` specifies if and how hard data is to be compressed. `complevel=0` and `complevel=None` disables compression and `0<complevel<10` enables compression.
- `complib` specifies which compression library to use. If nothing is specified the default library `zlib` is used. A compression library usually optimizes for either good compression rates or speed and the results will depend on the type of data. Which type of compression to choose depends on your specific needs and data. The list of supported compression libraries:
  - zlib: The default compression library. A classic in terms of compression, achieves good compression rates but is somewhat slow.
  - lzo: Fast compression and decompression.
  - bzip2: Good compression rates.
  - blosc: Fast compression and decompression.

    Support for alternative blosc compressors:

    - blosc:blosclz This is the default compressor for `blosc`
    - blosc:lz4: A compact, very popular and fast compressor.
    - blosc:lz4hc: A tweaked version of LZ4, produces better compression ratios at the expense of speed.
    - blosc:snappy: A popular compressor used in many places.

Skip to main content

- blosc:zlib: A classic; somewhat slower than the previous ones, but achieving better compression ratios.
- blosc:zstd: An extremely well balanced codec; it provides the best compression ratios among the others above, and at reasonably fast speed.

If `complib` is defined as something other than the listed libraries a `ValueError` exception is issued.

Note

If the library specified with the `complib` option is missing on your platform, compression defaults to `zlib` without further ado.

Enable compression for all objects within the file:

```
store_compressed = pd.HDFStore(
    "store_compressed.h5", complevel=9, complib="blosc:blosclz"
)
```

Or on-the-fly compression (this only applies to tables) in stores where compression is not enabled:

```
store.append("df", df, complib="zlib", complevel=5)
```

ptrepack

`PyTables` offers better write performance when tables are compressed after they are written, as opposed to turning on compression at the very beginning. You can use the supplied `PyTables` utility `ptrepack`. In addition, `ptrepack` can change compression levels after the fact.

```
ptrepack --chunkshape=auto --propindexes --complevel=9 --complib=blosc in.h5 out.h5
```

Furthermore `ptrepack in.h5 out.h5` will *repack* the file to allow you to reuse previously deleted space. Alternatively, one can simply remove the file and write again, or use the `copy` method.

Caveats

Skip to main content

Warning

`HDFStore` is **not-threadsafe for writing**. The underlying `PyTables` only supports concurrent reads (via threading or processes). If you need reading and writing *at the same time*, you need to serialize these operations in a single thread in a single process. You will corrupt your data otherwise. See the (GH 2397) for more information.

- If you use locks to manage write access between multiple processes, you may want to use `fsync()` before releasing write locks. For convenience you can use `store.flush(fsync=True)` to do this for you.
- Once a `table` is created columns (DataFrame) are fixed; only exactly the same columns can be appended
- Be aware that timezones (e.g., `pytz.timezone('US/Eastern')`) are not necessarily equal across timezone versions. So if data is localized to a specific timezone in the HDFStore using one version of a timezone library and that data is updated with another version, the data will be converted to UTC since these timezones are not considered equal. Either use the same version of timezone library or use `tz_convert` with the updated timezone definition.

Warning

`PyTables` will show a `NaturalNameWarning` if a column name cannot be used as an attribute selector. *Natural* identifiers contain only letters, numbers, and underscores, and may not begin with a number. Other identifiers cannot be used in a `where` clause and are generally a bad idea.

DataTypes

`HDFStore` will map an object dtype to the `PyTables` underlying dtype. This means the following types are known to work:

Type	Represents missing values
floating : `float64, float32, float16`	`np.nan`
integer : `int64, int32, int8, uint64,uint32, uint8`	
boolean	

Skip to main content

Type	Represents missing values
`timedelta64[ns]`	`NaT`
categorical : see the section below	
object : `strings`	`np.nan`

`unicode` columns are not supported, and **WILL FAIL**.

Categorical data

You can write data that contains `category` dtypes to a `HDFStore`. Queries work the same as if it was an object array. However, the `category` dtyped data is stored in a more efficient manner.

```
In [584]: dfcat = pd.DataFrame(
   .....:     {"A": pd.Series(list("aabbcdba")).astype("category"), "B": np.random.rand
   .....: )
   .....:

In [585]: dfcat
Out[585]:
   A         B
0  a -1.520478
1  a -1.069391
2  b -0.551981
3  b  0.452407
4  c  0.409257
5  d  0.301911
6  b -0.640843
7  a -2.253022

In [586]: dfcat.dtypes
Out[586]:
A    category
B     float64
dtype: object

In [587]: cstore = pd.HDFStore("cats.h5", mode="w")

In [588]: cstore.append("dfcat", dfcat, format="table", data_columns=["A"])

In [589]: result = cstore.select("dfcat", where="A in ['b', 'c']")

In [590]: result
Out[590]:
   A         B
2  b -0.551981
3  b  0.452407
4  c  0.409257
```

Skip to main content

```
In [591]: result.dtypes
Out[591]:
A    category
B     float64
dtype: object
```

String columns

min_itemsize

The underlying implementation of `HDFStore` uses a fixed column width (itemsize) for string columns. A string column itemsize is calculated as the maximum of the length of data (for that column) that is passed to the `HDFStore`, **in the first append**. Subsequent appends, may introduce a string for a column **larger** than the column can hold, an Exception will be raised (otherwise you could have a silent truncation of these columns, leading to loss of information). In the future we may relax this and allow a user-specified truncation to occur.

Pass `min_itemsize` on the first table creation to a-priori specify the minimum length of a particular string column. `min_itemsize` can be an integer, or a dict mapping a column name to an integer. You can pass `values` as a key to allow all *indexables* or *data_columns* to have this min_itemsize.

Passing a `min_itemsize` dict will cause all passed columns to be created as *data_columns* automatically.

> **Note**
>
> If you are not passing any `data_columns`, then the `min_itemsize` will be the maximum of the length of any string passed

```
In [592]: dfs = pd.DataFrame({"A": "foo", "B": "bar"}, index=list(range(5)))

In [593]: dfs
Out[593]:
     A    B
0  foo  bar
1  foo  bar
2  foo  bar
3  foo  bar
4  foo  bar

# A and B have a size of 30
In [594]: store.append("dfs", dfs, min_itemsize=30)
```

Skip to main content

```
In [595]: store.get_storer("dfs").table
Out[595]:
/dfs/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": StringCol(itemsize=30, shape=(2,), dflt=b'', pos=1)}
  byteorder := 'little'
  chunkshape := (963,)
  autoindex := True
  colindexes := {
    "index": Index(6, mediumshuffle, zlib(1)).is_csi=False}

# A is created as a data_column with a size of 30
# B is size is calculated
In [596]: store.append("dfs2", dfs, min_itemsize={"A": 30})

In [597]: store.get_storer("dfs2").table
Out[597]:
/dfs2/table (Table(5,)) ''
  description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": StringCol(itemsize=3, shape=(1,), dflt=b'', pos=1),
  "A": StringCol(itemsize=30, shape=(), dflt=b'', pos=2)}
  byteorder := 'little'
  chunkshape := (1598,)
  autoindex := True
  colindexes := {
    "index": Index(6, mediumshuffle, zlib(1)).is_csi=False,
    "A": Index(6, mediumshuffle, zlib(1)).is_csi=False}
```

nan_rep

String columns will serialize a `np.nan` (a missing value) with the `nan_rep` string representation. This defaults to the string value `nan`. You could inadvertently turn an actual `nan` value into a missing value.

```
In [598]: dfss = pd.DataFrame({"A": ["foo", "bar", "nan"]})

In [599]: dfss
Out[599]:
     A
0  foo
1  bar
2  nan

In [600]: store.append("dfss", dfss)

In [601]: store.select("dfss")
Out[601]:
     A
0  foo
1  bar
2  NaN
```

Skip to main content

```
In [602]: store.append("dfss2", dfss, nan_rep="_nan_")

In [603]: store.select("dfss2")
Out[603]:
     A
0  foo
1  bar
2  nan
```

Performance

- `tables` format come with a writing performance penalty as compared to `fixed` stores. The benefit is the ability to append/delete and query (potentially very large amounts of data). Write times are generally longer as compared with regular stores. Query times can be quite fast, especially on an indexed axis.
- You can pass `chunksize=<int>` to `append`, specifying the write chunksize (default is 50000). This will significantly lower your memory usage on writing.
- You can pass `expectedrows=<int>` to the first `append`, to set the TOTAL number of rows that `PyTables` will expect. This will optimize read/write performance.
- Duplicate rows can be written to tables, but are filtered out in selection (with the last items being selected; thus a table is unique on major, minor pairs)
- A `PerformanceWarning` will be raised if you are attempting to store types that will be pickled by PyTables (rather than stored as endemic types). See Here for more information and some solutions.

Feather

Feather provides binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy.

Feather is designed to faithfully serialize and de-serialize DataFrames, supporting all of the pandas dtypes, including extension dtypes such as categorical and datetime with tz.

Several caveats:

- The format will NOT write an `Index`, or `MultiIndex` for the `DataFrame` and will raise an error if a non-default one is provided. You can `.reset_index()` to store the index or `.reset_index(drop=True)` to ignore it.
- Duplicate column names and non-string columns names are not supported

Skip to main content

- Actual Python objects in object dtype columns are not supported. These will raise a helpful error message on an attempt at serialization.

See the Full Documentation.

```
In [604]: df = pd.DataFrame(
   .....:     {
   .....:         "a": list("abc"),
   .....:         "b": list(range(1, 4)),
   .....:         "c": np.arange(3, 6).astype("u1"),
   .....:         "d": np.arange(4.0, 7.0, dtype="float64"),
   .....:         "e": [True, False, True],
   .....:         "f": pd.Categorical(list("abc")),
   .....:         "g": pd.date_range("20130101", periods=3),
   .....:         "h": pd.date_range("20130101", periods=3, tz="US/Eastern"),
   .....:         "i": pd.date_range("20130101", periods=3, freq="ns"),
   .....:     }
   ..... )
   .....:

In [605]: df
Out[605]:
   a  b  c    d      e  f          g                         h
0  a  1  3  4.0   True  a 2013-01-01 2013-01-01 00:00:00-05:00 2013-01-01 00:00:00.0000
1  b  2  4  5.0  False  b 2013-01-02 2013-01-02 00:00:00-05:00 2013-01-01 00:00:00.0000
2  c  3  5  6.0   True  c 2013-01-03 2013-01-03 00:00:00-05:00 2013-01-01 00:00:00.0000

In [606]: df.dtypes
Out[606]:
a                          object
b                           int64
c                           uint8
d                         float64
e                            bool
f                        category
g                  datetime64[ns]
h    datetime64[ns, US/Eastern]
i                  datetime64[ns]
dtype: object
```

Write to a feather file.

```
In [607]: df.to_feather("example.feather")
```

Read from a feather file.

```
In [608]: result = pd.read_feather("example.feather")

In [609]: result
```

Skip to main content

```
0  a  1  3  4.0   True  a 2013-01-01 2013-01-01 00:00:00-05:00 2013-01-01 00:00:00.0000
1  b  2  4  5.0  False  b 2013-01-02 2013-01-02 00:00:00-05:00 2013-01-01 00:00:00.0000
2  c  3  5  6.0   True  c 2013-01-03 2013-01-03 00:00:00-05:00 2013-01-01 00:00:00.0000

# we preserve dtypes
In [610]: result.dtypes
Out[610]:
a                          object
b                           int64
c                           uint8
d                         float64
e                            bool
f                        category
g                  datetime64[ns]
h    datetime64[ns, US/Eastern]
i                  datetime64[ns]
dtype: object
```

Parquet

[Apache Parquet](#) provides a partitioned binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy. Parquet can use a variety of compression techniques to shrink the file size as much as possible while still maintaining good read performance.

Parquet is designed to faithfully serialize and de-serialize `DataFrame` s, supporting all of the pandas dtypes, including extension dtypes such as datetime with tz.

Several caveats.

- Duplicate column names and non-string columns names are not supported.
- The `pyarrow` engine always writes the index to the output, but `fastparquet` only writes non-default indexes. This extra column can cause problems for non-pandas consumers that are not expecting it. You can force including or omitting indexes with the `index` argument, regardless of the underlying engine.
- Index level names, if specified, must be strings.
- In the `pyarrow` engine, categorical dtypes for non-string types can be serialized to parquet, but will de-serialize as their primitive dtype.
- The `pyarrow` engine preserves the `ordered` flag of categorical dtypes with string types. `fastparquet` does not preserve the `ordered` flag.
- Non supported types include `Interval` and actual Python object types. These will raise a helpful error message on an attempt at serialization. `Period` type is supported with pyarrow

Skip to main content

- The `pyarrow` engine preserves extension data types such as the nullable integer and string data type (requiring pyarrow >= 0.16.0, and requiring the extension type to implement the needed protocols, see the extension types documentation).

You can specify an `engine` to direct the serialization. This can be one of `pyarrow`, or `fastparquet`, or `auto`. If the engine is NOT specified, then the `pd.options.io.parquet.engine` option is checked; if this is also `auto`, then `pyarrow` is tried, and falling back to `fastparquet`.

See the documentation for pyarrow and fastparquet.

Note

These engines are very similar and should read/write nearly identical parquet format files. `pyarrow>=8.0.0` supports timedelta data, `fastparquet>=0.1.4` supports timezone aware datetimes. These libraries differ by having different underlying dependencies (`fastparquet` by using `numba`, while `pyarrow` uses a c-library).

```
In [611]: df = pd.DataFrame(
   .....:     {
   .....:         "a": list("abc"),
   .....:         "b": list(range(1, 4)),
   .....:         "c": np.arange(3, 6).astype("u1"),
   .....:         "d": np.arange(4.0, 7.0, dtype="float64"),
   .....:         "e": [True, False, True],
   .....:         "f": pd.date_range("20130101", periods=3),
   .....:         "g": pd.date_range("20130101", periods=3, tz="US/Eastern"),
   .....:         "h": pd.Categorical(list("abc")),
   .....:         "i": pd.Categorical(list("abc"), ordered=True),
   .....:     }
   ..... : )
   .....:

In [612]: df
Out[612]:
   a  b  c    d      e          f                         g  h  i
0  a  1  3  4.0   True 2013-01-01 2013-01-01 00:00:00-05:00  a  a
1  b  2  4  5.0  False 2013-01-02 2013-01-02 00:00:00-05:00  b  b
2  c  3  5  6.0   True 2013-01-03 2013-01-03 00:00:00-05:00  c  c

In [613]: df.dtypes
Out[613]:
a                       object
b                        int64
c                        uint8
d                      float64
e                         bool
f                 datetime64[ns]
g    datetime64[ns, US/Eastern]
h                     category
```

Skip to main content

```
i                     category
dtype: object
```

Write to a parquet file.

```
In [614]: df.to_parquet("example_pa.parquet", engine="pyarrow")

In [615]: df.to_parquet("example_fp.parquet", engine="fastparquet")
```

Read from a parquet file.

```
In [616]: result = pd.read_parquet("example_fp.parquet", engine="fastparquet")

In [617]: result = pd.read_parquet("example_pa.parquet", engine="pyarrow")

In [618]: result.dtypes
Out[618]:
a                       object
b                        int64
c                        uint8
d                      float64
e                         bool
f              datetime64[ns]
g    datetime64[ns, US/Eastern]
h                     category
i                     category
dtype: object
```

By setting the `dtype_backend` argument you can control the default dtypes used for the resulting DataFrame.

```
In [619]: result = pd.read_parquet("example_pa.parquet", engine="pyarrow", dtype_backen

In [620]: result.dtypes
Out[620]:
a                                    string[pyarrow]
b                                     int64[pyarrow]
c                                     uint8[pyarrow]
d                                    double[pyarrow]
e                                      bool[pyarrow]
f                             timestamp[ns][pyarrow]
g              timestamp[ns, tz=US/Eastern][pyarrow]
h    dictionary<values=string, indices=int32, order...
i    dictionary<values=string, indices=int32, order...
dtype: object
```

Skip to main content

Note

Note that this is not supported for `fastparquet`.

Read only certain columns of a parquet file.

```
In [621]: result = pd.read_parquet(
   .....:     "example_fp.parquet",
   .....:     engine="fastparquet",
   .....:     columns=["a", "b"],
   .....: )
   .....:

In [622]: result = pd.read_parquet(
   .....:     "example_pa.parquet",
   .....:     engine="pyarrow",
   .....:     columns=["a", "b"],
   .....: )
   .....:

In [623]: result.dtypes
Out[623]:
a    object
b     int64
dtype: object
```

Handling indexes

Serializing a `DataFrame` to parquet may include the implicit index as one or more columns in the output file. Thus, this code:

```
In [624]: df = pd.DataFrame({"a": [1, 2], "b": [3, 4]})

In [625]: df.to_parquet("test.parquet", engine="pyarrow")
```

creates a parquet file with *three* columns if you use `pyarrow` for serialization: `a`, `b`, and `__index_level_0__`. If you're using `fastparquet`, the index may or may not be written to the file.

This unexpected extra column causes some databases like Amazon Redshift to reject the file, because that column doesn't exist in the target table.

If you want to omit a dataframe's indexes when writing, pass `index=False` to `to_parquet()`:

Skip to main content

```
In [626]: df.to_parquet("test.parquet", index=False)
```

This creates a parquet file with just the two expected columns, `a` and `b`. If your `DataFrame` has a custom index, you won't get it back when you load this file into a `DataFrame`.

Passing `index=True` will *always* write the index, even if that's not the underlying engine's default behavior.

Partitioning Parquet files

Parquet supports partitioning of data based on the values of one or more columns.

```
In [627]: df = pd.DataFrame({"a": [0, 0, 1, 1], "b": [0, 1, 0, 1]})

In [628]: df.to_parquet(path="test", engine="pyarrow", partition_cols=["a"], compressio
```

The `path` specifies the parent directory to which data will be saved. The `partition_cols` are the column names by which the dataset will be partitioned. Columns are partitioned in the order they are given. The partition splits are determined by the unique values in the partition columns. The above example creates a partitioned dataset that may look like:

```
test
├── a=0
│   ├── 0bac803e32dc42ae83fddfd029cbdebc.parquet
│   └──  ...
└── a=1
    ├── e6ab24a4f45147b49b54a662f0c412a3.parquet
    └── ...
```

ORC

Similar to the parquet format, the ORC Format is a binary columnar serialization for data frames. It is designed to make reading data frames efficient. pandas provides both the reader and the writer for the ORC format, `read_orc()` and `to_orc()`. This requires the pyarrow library.

Skip to main content

Warning

- It is *highly recommended* to install pyarrow using conda due to some issues occurred by pyarrow.
- `to_orc()` requires pyarrow>=7.0.0.
- `read_orc()` and `to_orc()` are not supported on Windows yet, you can find valid environments on install optional dependencies.
- For supported dtypes please refer to supported ORC features in Arrow.
- Currently timezones in datetime columns are not preserved when a dataframe is converted into ORC files.

```
In [629]: df = pd.DataFrame(
   .....:     {
   .....:         "a": list("abc"),
   .....:         "b": list(range(1, 4)),
   .....:         "c": np.arange(4.0, 7.0, dtype="float64"),
   .....:         "d": [True, False, True],
   .....:         "e": pd.date_range("20130101", periods=3),
   .....:     }
   .....: )
   .....:

In [630]: df
Out[630]:
   a  b    c      d          e
0  a  1  4.0   True 2013-01-01
1  b  2  5.0  False 2013-01-02
2  c  3  6.0   True 2013-01-03

In [631]: df.dtypes
Out[631]:
a            object
b             int64
c           float64
d              bool
e    datetime64[ns]
dtype: object
```

Write to an orc file.

```
In [632]: df.to_orc("example_pa.orc", engine="pyarrow")
```

Read from an orc file.

```
In [633]: result = pd.read_orc("example_pa.orc")
```

Skip to main content

```
a            object
b             int64
c           float64
d              bool
e    datetime64[ns]
dtype: object
```

Read only certain columns of an orc file.

```
In [635]: result = pd.read_orc(
   .....:     "example_pa.orc",
   .....:     columns=["a", "b"],
   .....: )
   .....:

In [636]: result.dtypes
Out[636]:
a    object
b     int64
dtype: object
```

SQL queries

The `pandas.io.sql` module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API.

Where available, users may first want to opt for Apache Arrow ADBC drivers. These drivers should provide the best performance, null handling, and type detection.

> **New in version 2.2.0:** Added native support for ADBC drivers

For a full list of ADBC drivers and their development status, see the ADBC Driver Implementation Status documentation.

Where an ADBC driver is not available or may be missing functionality, users should opt for installing SQLAlchemy alongside their database driver library. Examples of such drivers are psycopg2 for PostgreSQL or pymysql for MySQL. For SQLite this is included in Python's standard library by default. You can find an overview of supported drivers for each SQL dialect in the SQLAlchemy docs.

Skip to main content

If SQLAlchemy is not installed, you can use a `sqlite3.Connection` in place of a SQLAlchemy engine, connection, or URI string.

See also some cookbook examples for some advanced strategies.

The key functions are:

`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.
`DataFrame.to_sql`(name, con, *[, schema, ...])	Write records stored in a DataFrame to a SQL database.

Note

The function `read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()` (and for backward compatibility) and will delegate to specific function depending on the provided input (database table name or sql query). Table names do not need to be quoted if they have special characters.

In the following example, we use the SQlite SQL database engine. You can use a temporary SQLite database where data are stored in "memory".

To connect using an ADBC driver you will want to install the `adbc_driver_sqlite` using your package manager. Once installed, you can use the DBAPI interface provided by the ADBC driver to connect to your database.

```
import adbc_driver_sqlite.dbapi as sqlite_dbapi

# Create the connection
with sqlite_dbapi.connect("sqlite:///:memory:") as conn:
     df = pd.read_sql_table("data", conn)
```

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to. For more information on `create_engine()` and the URI formatting, see the examples below and the SQLAlchemy documentation

Skip to main content

```
In [637]: from sqlalchemy import create_engine

# Create your engine.
In [638]: engine = create_engine("sqlite:///:memory:")
```

If you want to manage your own connections you can pass one of those instead. The example below opens a connection to the database using a Python context manager that automatically closes the connection after the block has completed. See the SQLAlchemy docs for an explanation of how the database connection is handled.

```
with engine.connect() as conn, conn.begin():
    data = pd.read_sql_table("data", conn)
```

Warning

When you open a connection to a database you are also responsible for closing it. Side effects of leaving a connection open may include locking the database or other breaking behaviour.

Writing DataFrames

Assuming the following data is in a `DataFrame` `data`, we can insert it into the database using `to_sql()`.

id	Date	Col_1	Col_2	Col_3
26	2012-10-18	X	25.7	True
42	2012-10-19	Y	-12.4	False
63	2012-10-20	Z	5.73	True

```
In [639]: import datetime

In [640]: c = ["id", "Date", "Col_1", "Col_2", "Col_3"]

In [641]: d = [
   .....:     (26, datetime.datetime(2010, 10, 18), "X", 27.5, True),
   .....:     (42, datetime.datetime(2010, 10, 19), "Y", -12.5, False),
   .....:     (63, datetime.datetime(2010, 10, 20), "Z", 5.73, True),
   .....: ]
```

Skip to main content

```
In [642]: data = pd.DataFrame(d, columns=c)

In [643]: data
Out[643]:
   id       Date Col_1  Col_2  Col_3
0  26 2010-10-18     X  27.50   True
1  42 2010-10-19     Y -12.50  False
2  63 2010-10-20     Z   5.73   True

In [644]: data.to_sql("data", con=engine)
Out[644]: 3
```

With some databases, writing large DataFrames can result in errors due to packet size limitations being exceeded. This can be avoided by setting the `chunksize` parameter when calling `to_sql`. For example, the following writes `data` to the database in batches of 1000 rows at a time:

```
In [645]: data.to_sql("data_chunked", con=engine, chunksize=1000)
Out[645]: 3
```

SQL data types

Ensuring consistent data type management across SQL databases is challenging. Not every SQL database offers the same types, and even when they do the implementation of a given type can vary in ways that have subtle effects on how types can be preserved.

For the best odds at preserving database types users are advised to use ADBC drivers when available. The Arrow type system offers a wider array of types that more closely match database types than the historical pandas/NumPy type system. To illustrate, note this (non-exhaustive) listing of types available in different databases and pandas backends:

numpy/pandas	arrow	postgres	sqlite
int16/Int16	int16	SMALLINT	INTEGER
int32/Int32	int32	INTEGER	INTEGER
int64/Int64	int64	BIGINT	INTEGER
float32	float32	REAL	REAL
float64	float64	DOUBLE PRECISION	REAL
object	string	TEXT	TEXT

Skip to main content

numpy/pandas	arrow	postgres	sqlite
bool	`bool_`	BOOLEAN	
datetime64[ns]	timestamp(us)	TIMESTAMP	
datetime64[ns,tz]	timestamp(us,tz)	TIMESTAMPTZ	
	date32	DATE	
	month_day_nano_interval	INTERVAL	
	binary	BINARY	BLOB
	decimal128	DECIMAL [1]	
	list	ARRAY [1]	
	struct	**COMPOSITE TYPE** [1]	

Footnotes

[1](1,2,3) Not implemented as of writing, but theoretically possible

If you are interested in preserving database types as best as possible throughout the lifecycle of your DataFrame, users are encouraged to leverage the `dtype_backend="pyarrow"` argument of `read_sql()`

```
# for roundtripping
with pg_dbapi.connect(uri) as conn:
    df2 = pd.read_sql("pandas_table", conn, dtype_backend="pyarrow")
```

This will prevent your data from being converted to the traditional pandas/NumPy type system, which often converts SQL types in ways that make them impossible to round-trip.

In case an ADBC driver is not available, `to_sql()` will try to map your data to an appropriate SQL data type based on the dtype of the data. When you have columns of dtype `object`, pandas will try to infer the data type.

You can always override the default type by specifying the desired SQL type of any of the columns by using the `dtype` argument. This argument needs a dictionary mapping column

Skip to main content

names to SQLAlchemy types (or strings for the sqlite3 fallback mode). For example, specifying to use the sqlalchemy `String` type instead of the default `Text` type for string columns:

```
In [646]: from sqlalchemy.types import String

In [647]: data.to_sql("data_dtype", con=engine, dtype={"Col_1": String})
Out[647]: 3
```

Note

Due to the limited support for timedelta's in the different database flavors, columns with type `timedelta64` will be written as integer values as nanoseconds to the database and a warning will be raised. The only exception to this is when using the ADBC PostgreSQL driver in which case a timedelta will be written to the database as an `INTERVAL`

Note

Columns of `category` dtype will be converted to the dense representation as you would get with `np.asarray(categorical)` (e.g. for string categories this gives an array of strings). Because of this, reading the database table back in does **not** generate a categorical.

Datetime data types

Using ADBC or SQLAlchemy, `to_sql()` is capable of writing datetime data that is timezone naive or timezone aware. However, the resulting data stored in the database ultimately depends on the supported data type for datetime data of the database system being used.

The following table lists supported data types for datetime data for some common databases. Other database dialects may have different data types for datetime data.

Database	SQL Datetime Types	Timezone Support
SQLite	`TEXT`	No
MySQL	`TIMESTAMP` or `DATETIME`	No
PostgreSQL	`TIMESTAMP` or `TIMESTAMP WITH TIME ZONE`	Yes

Skip to main content

When writing timezone aware data to databases that do not support timezones, the data will be written as timezone naive timestamps that are in local time with respect to the timezone.

`read_sql_table()` is also capable of reading datetime data that is timezone aware or naive. When reading `TIMESTAMP WITH TIME ZONE` types, pandas will convert the data to UTC.

Insertion method

The parameter `method` controls the SQL insertion clause used. Possible values are:

- `None`: Uses standard SQL `INSERT` clause (one per row).
- `'multi'`: Pass multiple values in a single `INSERT` clause. It uses a *special* SQL syntax not supported by all backends. This usually provides better performance for analytic databases like *Presto* and *Redshift*, but has worse performance for traditional SQL backend if the table contains many columns. For more information check the SQLAlchemy documentation.
- callable with signature `(pd_table, conn, keys, data_iter)`: This can be used to implement a more performant insertion method based on specific backend dialect features.

Example of a callable using PostgreSQL COPY clause:

```
# Alternative to_sql() *method* for DBs that support COPY FROM
import csv
from io import StringIO

def psql_insert_copy(table, conn, keys, data_iter):
    """
    Execute SQL statement inserting data

    Parameters
    ----------
    table : pandas.io.sql.SQLTable
    conn : sqlalchemy.engine.Engine or sqlalchemy.engine.Connection
    keys : list of str
        Column names
    data_iter : Iterable that iterates the values to be inserted
    """
    # gets a DBAPI connection that can provide a cursor
    dbapi_conn = conn.connection
    with dbapi_conn.cursor() as cur:
        s_buf = StringIO()
        writer = csv.writer(s_buf)
        writer.writerows(data_iter)
        s_buf.seek(0)

        columns = ', '.join(['"{}"'.format(k) for k in keys])
        if table.schema:
            table_name = '{}.{}'.format(table.schema, table.name)
```

Skip to main content

```
        sql = 'COPY {} ({}) FROM STDIN WITH CSV'.format(
            table_name, columns)
        cur.copy_expert(sql=sql, file=s_buf)
```

Reading tables

`read_sql_table()` will read a database table given the table name and optionally a subset of columns to read.

Note

In order to use `read_sql_table()`, you **must** have the ADBC driver or SQLAlchemy optional dependency installed.

```
In [648]: pd.read_sql_table("data", engine)
Out[648]:
   index  id       Date Col_1  Col_2  Col_3
0      0  26 2010-10-18     X  27.50   True
1      1  42 2010-10-19     Y -12.50  False
2      2  63 2010-10-20     Z   5.73   True
```

Note

ADBC drivers will map database types directly back to arrow types. For other drivers note that pandas infers column dtypes from query outputs, and not by looking up data types in the physical database schema. For example, assume `userid` is an integer column in a table. Then, intuitively, `select userid ...` will return integer-valued series, while `select cast(userid as text) ...` will return object-valued (str) series. Accordingly, if the query output is empty, then all resulting columns will be returned as object-valued (since they are most general). If you foresee that your query will sometimes generate an empty result, you may want to explicitly typecast afterwards to ensure dtype integrity.

You can also specify the name of the column as the `DataFrame` index, and specify a subset of columns to be read.

```
In [649]: pd.read_sql_table("data", engine, index_col="id")
Out[649]:
```

Skip to main content

```
26      0 2010-10-18     X  27.50   True
42      1 2010-10-19     Y -12.50  False
63      2 2010-10-20     Z   5.73   True

In [650]: pd.read_sql_table("data", engine, columns=["Col_1", "Col_2"])
Out[650]:
  Col_1  Col_2
0     X  27.50
1     Y -12.50
2     Z   5.73
```

And you can explicitly force columns to be parsed as dates:

```
In [651]: pd.read_sql_table("data", engine, parse_dates=["Date"])
Out[651]:
   index  id       Date Col_1  Col_2  Col_3
0      0  26 2010-10-18     X  27.50   True
1      1  42 2010-10-19     Y -12.50  False
2      2  63 2010-10-20     Z   5.73   True
```

If needed you can explicitly specify a format string, or a dict of arguments to pass to `pandas.to_datetime()`:

```
pd.read_sql_table("data", engine, parse_dates={"Date": "%Y-%m-%d"})
pd.read_sql_table(
    "data",
    engine,
    parse_dates={"Date": {"format": "%Y-%m-%d %H:%M:%S"}},
)
```

You can check if a table exists using `has_table()`

Schema support

Reading from and writing to different schema's is supported through the `schema` keyword in the `read_sql_table()` and `to_sql()` functions. Note however that this depends on the database flavor (sqlite does not have schema's). For example:

```
df.to_sql(name="table", con=engine, schema="other_schema")
pd.read_sql_table("table", engine, schema="other_schema")
```

Querying

Skip to main content

You can query using raw SQL in the `read_sql_query()` function. In this case you must use the SQL variant appropriate for your database. When using SQLAlchemy, you can also pass SQLAlchemy Expression language constructs, which are database-agnostic.

```
In [652]: pd.read_sql_query("SELECT * FROM data", engine)
Out[652]:
   index  id                        Date Col_1  Col_2  Col_3
0      0  26  2010-10-18 00:00:00.000000     X  27.50      1
1      1  42  2010-10-19 00:00:00.000000     Y -12.50      0
2      2  63  2010-10-20 00:00:00.000000     Z   5.73      1
```

Of course, you can specify a more “complex” query.

```
In [653]: pd.read_sql_query("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;", engine)
Out[653]:
   id Col_1  Col_2
0  42     Y  -12.5
```

The `read_sql_query()` function supports a `chunksize` argument. Specifying this will return an iterator through chunks of the query result:

```
In [654]: df = pd.DataFrame(np.random.randn(20, 3), columns=list("abc"))

In [655]: df.to_sql(name="data_chunks", con=engine, index=False)
Out[655]: 20
```

```
In [656]: for chunk in pd.read_sql_query("SELECT * FROM data_chunks", engine, chunksize
   .....:     print(chunk)
   .....:
          a         b         c
0 -0.395347 -0.822726 -0.363777
1  1.676124 -0.908102 -1.391346
2 -1.094269  0.278380  1.205899
3  1.503443  0.932171 -0.709459
4 -0.645944 -1.351389  0.132023
          a         b         c
0  0.210427  0.192202  0.661949
1  1.690629 -1.046044  0.618697
2 -0.013863  1.314289  1.951611
3 -1.485026  0.304662  1.194757
4 -0.446717  0.528496 -0.657575
          a         b         c
0 -0.876654  0.336252  0.172668
1  0.337684 -0.411202 -0.828394
2 -0.244413  1.094948  0.087183
3  1.125934 -1.480095  1.205944
4 -0.451849  0.452214 -2.208192
```

Skip to main content

```
1  1.248959 -0.675595 -1.908296
2 -0.125934  1.491974  0.648726
3  0.391214  0.438609  1.634248
4  1.208707 -1.535740  1.620399
```

Engine connection examples

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to.

```
from sqlalchemy import create_engine

engine = create_engine("postgresql://scott:tiger@localhost:5432/mydatabase")

engine = create_engine("mysql+mysqldb://scott:tiger@localhost/foo")

engine = create_engine("oracle://scott:tiger@127.0.0.1:1521/sidname")

engine = create_engine("mssql+pyodbc://mydsn")

# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine("sqlite:///foo.db")

# or absolute, starting with a slash:
engine = create_engine("sqlite:////absolute/path/to/foo.db")
```

For more information see the examples the SQLAlchemy documentation

Advanced SQLAlchemy queries

You can use SQLAlchemy constructs to describe your query.

Use `sqlalchemy.text()` to specify query parameters in a backend-neutral way

```
In [657]: import sqlalchemy as sa

In [658]: pd.read_sql(
   .....:     sa.text("SELECT * FROM data where Col_1=:col1"), engine, params={"col1":
   .....: )
   .....:
Out[658]:
   index  id                        Date Col_1  Col_2  Col_3
0      0  26  2010-10-18 00:00:00.000000     X   27.5      1
```

Skip to main content

If you have an SQLAlchemy description of your database you can express where conditions using SQLAlchemy expressions

```
In [659]: metadata = sa.MetaData()

In [660]: data_table = sa.Table(
   .....:     "data",
   .....:     metadata,
   .....:     sa.Column("index", sa.Integer),
   .....:     sa.Column("Date", sa.DateTime),
   .....:     sa.Column("Col_1", sa.String),
   .....:     sa.Column("Col_2", sa.Float),
   .....:     sa.Column("Col_3", sa.Boolean),
   .....: )
   .....:

In [661]: pd.read_sql(sa.select(data_table).where(data_table.c.Col_3 is True), engine)
Out[661]:
Empty DataFrame
Columns: [index, Date, Col_1, Col_2, Col_3]
Index: []
```

You can combine SQLAlchemy expressions with parameters passed to `read_sql()` using `sqlalchemy.bindparam()`

```
In [662]: import datetime as dt

In [663]: expr = sa.select(data_table).where(data_table.c.Date > sa.bindparam("date"))

In [664]: pd.read_sql(expr, engine, params={"date": dt.datetime(2010, 10, 18)})
Out[664]:
   index       Date Col_1  Col_2  Col_3
0      1 2010-10-19     Y -12.50  False
1      2 2010-10-20     Z   5.73   True
```

Sqlite fallback

The use of sqlite is supported without using SQLAlchemy. This mode requires a Python database adapter which respect the Python DB-API.

You can create connections like so:

```
import sqlite3

con = sqlite3.connect(":memory:")
```

Skip to main content

And then issue the following queries:

```
data.to_sql("data", con)
pd.read_sql_query("SELECT * FROM data", con)
```

Google BigQuery

The `pandas-gbq` package provides functionality to read/write from Google BigQuery.

pandas integrates with this external package. if `pandas-gbq` is installed, you can use the pandas methods `pd.read_gbq` and `DataFrame.to_gbq`, which will call the respective functions from `pandas-gbq`.

Full documentation can be found here.

Stata format

Writing to stata format

The method `DataFrame.to_stata()` will write a DataFrame into a .dta file. The format version of this file is always 115 (Stata 12).

```
In [665]: df = pd.DataFrame(np.random.randn(10, 2), columns=list("AB"))

In [666]: df.to_stata("stata.dta")
```

Stata data files have limited data type support; only strings with 244 or fewer characters, `int8`, `int16`, `int32`, `float32` and `float64` can be stored in `.dta` files. Additionally, *Stata* reserves certain values to represent missing data. Exporting a non-missing value that is outside of the permitted range in Stata for a particular data type will retype the variable to the next larger size. For example, `int8` values are restricted to lie between -127 and 100 in Stata, and so variables with values above 100 will trigger a conversion to `int16`. `nan` values in floating points data types are stored as the basic missing data type (`.` in *Stata*).

> **Note**
>
> It is not possible to export missing data values for integer data types.

Skip to main content

The *Stata* writer gracefully handles other data types including `int64`, `bool`, `uint8`, `uint16`, `uint32` by casting to the smallest supported type that can represent the data. For example, data with a type of `uint8` will be cast to `int8` if all values are less than 100 (the upper bound for non-missing `int8` data in *Stata*), or, if values are outside of this range, the variable is cast to `int16`.

Warning

Conversion from `int64` to `float64` may result in a loss of precision if `int64` values are larger than 2**53.

Warning

`StataWriter` and `DataFrame.to_stata()` only support fixed width strings containing up to 244 characters, a limitation imposed by the version 115 dta file format. Attempting to write *Stata* dta files with strings longer than 244 characters raises a `ValueError`.

Reading from Stata format

The top-level function `read_stata` will read a dta file and return either a `DataFrame` or a `pandas.api.typing.StataReader` that can be used to read the file incrementally.

```
In [667]: pd.read_stata("stata.dta")
Out[667]:
   index         A         B
0      0 -0.165614  0.490482
1      1 -0.637829  0.067091
2      2 -0.242577  1.348038
3      3  0.647699 -0.644937
4      4  0.625771  0.918376
5      5  0.401781 -1.488919
6      6 -0.981845 -0.046882
7      7 -0.306796  0.877025
8      8 -0.336606  0.624747
9      9 -1.582600  0.806340
```

Specifying a `chunksize` yields a `pandas.api.typing.StataReader` instance that can be used to read `chunksize` lines from the file at a time. The `StataReader` object can be used as an iterator.

```
In [668]: with pd.read_stata("stata.dta", chunksize=3) as reader:
```

Skip to main content

```
   .....:
(3, 3)
(3, 3)
(3, 3)
(1, 3)
```

For more fine-grained control, use `iterator=True` and specify `chunksize` with each call to `read()`.

```
In [669]: with pd.read_stata("stata.dta", iterator=True) as reader:
   .....:     chunk1 = reader.read(5)
   .....:     chunk2 = reader.read(5)
   .....:
```

Currently the `index` is retrieved as a column.

The parameter `convert_categoricals` indicates whether value labels should be read and used to create a `Categorical` variable from them. Value labels can also be retrieved by the function `value_labels`, which requires `read()` to be called before use.

The parameter `convert_missing` indicates whether missing value representations in Stata should be preserved. If `False` (the default), missing values are represented as `np.nan`. If `True`, missing values are represented using `StataMissingValue` objects, and columns containing missing values will have `object` data type.

Note

`read_stata()` and `StataReader` support .dta formats 113-115 (Stata 10-12), 117 (Stata 13), and 118 (Stata 14).

Note

Setting `preserve_dtypes=False` will upcast to the standard pandas data types: `int64` for all integer types and `float64` for floating point data. By default, the Stata data types are preserved when importing.

Skip to main content

Note

All `StataReader` objects, whether created by `read_stata()` (when using `iterator=True` or `chunksize`) or instantiated by hand, must be used as context managers (e.g. the `with` statement). While the `close()` method is available, its use is unsupported. It is not part of the public API and will be removed in with future without warning.

Categorical data

`Categorical` data can be exported to *Stata* data files as value labeled data. The exported data consists of the underlying category codes as integer data values and the categories as value labels. *Stata* does not have an explicit equivalent to a `Categorical` and information about *whether* the variable is ordered is lost when exporting.

Warning

Stata only supports string value labels, and so `str` is called on the categories when exporting data. Exporting `Categorical` variables with non-string categories produces a warning, and can result a loss of information if the `str` representations of the categories are not unique.

Labeled data can similarly be imported from *Stata* data files as `Categorical` variables using the keyword argument `convert_categoricals` (`True` by default). The keyword argument `order_categoricals` (`True` by default) determines whether imported `Categorical` variables are ordered.

Skip to main content

Note

When importing categorical data, the values of the variables in the *Stata* data file are not preserved since `Categorical` variables always use integer data types between `-1` and `n-1` where `n` is the number of categories. If the original values in the *Stata* data file are required, these can be imported by setting `convert_categoricals=False`, which will import original data (but not the variable labels). The original values can be matched to the imported categorical data since there is a simple mapping between the original *Stata* data values and the category codes of imported Categorical variables: missing values are assigned code `-1`, and the smallest original value is assigned `0`, the second smallest is assigned `1` and so on until the largest original value is assigned the code `n-1`.

Note

Stata supports partially labeled series. These series have value labels for some but not all data values. Importing a partially labeled series will produce a `Categorical` with string categories for the values that are labeled and numeric categories for values with no label.

Previous
Essential basic functionality
Next
PyArrow Functionality