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	JSON	read_json	to_json
text	HTML	read_html	to_html
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	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	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 according to your Python version, i.e. `from StringIO import StringIO` for Python 2 and `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._path.local.LocalPath`**), URL (including http, ftp, and S3 locations), or any object with a `read()` method (such as an open file or **`StringIO`**).

Scroll To Top

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 character and different from `'\s+'` will be interpreted as regular expressions and will also 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.

New in version 0.18.1: support for the Python parser.

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, 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.

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). 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]: from io import StringIO, BytesIO

In [2]: data = ('col1,col2,col3\n'
   ...:         'a,b,1\n'
   ...:         'a,b,2\n'
   ...:         'c,d,3')
   ...:

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

In [4]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['COL1', 'COL3']
Out[4]:
  col1  col3
0    a     1
1    a     2
2    c     3
```

Using this parameter results in much faster parsing time and lower memory usage.

squeeze : *boolean, default* `False`
: If the parsed data only contains one column then return a `Series`.

prefix : *str, default* `None`
: Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, …

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

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}` (unsupported with `engine='python'`). Use *str* or *object* together with suitable `na_values` settings to preserve and not interpret dtype.

New in version 0.20.0: support for the Python parser.

engine : *{*`'c'`*,* `'python'`*}*
: Parser engine to use. The C engine is faster while the Python engine is currently more feature-complete.

converters : *dict, default* `None`
: Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

Scroll To Top

true_values : *list, default* `None`

Values to consider as `True`.

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 [5]: data = ('col1,col2,col3\n'
   ...:         'a,b,1\n'
   ...:         'a,b,2\n'
   ...:         'c,d,3')
   ...:

In [6]: pd.read_csv(StringIO(data))
Out[6]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [7]: pd.read_csv(StringIO(data), skiprows=lambda x: x % 2 != 0)
Out[7]:
  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.

NA and missing data handling

Scroll To Top

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'. 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.

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.

Scroll To Top

dayfirst : *boolean, default* `False`

DD/MM format dates, international and European format.

cache_dates : *boolean, default True*
: 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.

  New in version 0.25.0.

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

  New in version 0.18.1: support for 'zip' and 'xz' compression.

  Changed in version 0.24.0: 'infer' option added and set to default.

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 `None` for the ordinary converter, `high` for the high-precision converter, and `round_trip` for 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).

Scroll To Top

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

error_bad_lines : *boolean, default* `True`
: Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no `DataFrame` will be returned. If `False`, then these "bad lines" will dropped from the `DataFrame` that is returned. See bad lines below.

warn_bad_lines : *boolean, default* `True`
: If error_bad_lines is `False`, and warn_bad_lines is `True`, a warning for each "bad line" will be output.

Specifying column data types

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

```
In [8]: data = ('a,b,c,d\n'
   ...:         '1,2,3,4\n'
   ...:         '5,6,7,8\n'
   ...:         '9,10,11')
   ...:

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

In [10]: df = pd.read_csv(StringIO(data), dtype=object)

In [11]: df
Out[11]:
   a  b  c   d
0  1  2  3   4
```

Scroll To Top

```
1  5   6   7     8
2  9  10  11   NaN

In [12]: df['a'][0]
Out[12]: '1'

In [13]: df = pd.read_csv(StringIO(data),
   ....:                  dtype={'b': object, 'c': np.float64, 'd': 'Int64'})
   ....:

In [14]: df.dtypes
Out[14]:
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 `dtype`. If you're unfamiliar with these concepts, you can see here to learn more about dtypes, and here to learn more about `object` conversion in pandas.

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

```
In [15]: data = ("col_1\n"
   ....:         "1\n"
   ....:         "2\n"
   ....:         "'A'\n"
   ....:         "4.22")
   ....:

In [16]: df = pd.read_csv(StringIO(data), converters={'col_1': str})

In [17]: df
Out[17]:
  col_1
0     1
1     2
2   'A'
3  4.22

In [18]: df['col_1'].apply(type).value_counts()
Out[18]:
<class 'str'>    4
Name: col_1, dtype: int64
```

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

```
In [19]: df2 = pd.read_csv(StringIO(data))

In [20]: df2['col_1'] = pd.to_numeric(df2['col_1'], errors='coerce')

In [21]: df2
Out[21]:
   col_1
0   1.00
1   2.00
2    NaN
3   4.22
```

Scroll To Top

```
In [22]: df2['col_1'].apply(type).value_counts()
Out[22]:
<class 'float'>    4
Name: col_1, 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.

> *New in version 0.20.0:* support for the Python parser.
>
> The `dtype` option is supported by the 'python' engine.

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 [23]: col_1 = list(range(500000)) + ['a', 'b'] + list(range(500000))

In [24]: df = pd.DataFrame({'col_1': col_1})

In [25]: df.to_csv('foo.csv')

In [26]: mixed_df = pd.read_csv('foo.csv')

In [27]: mixed_df['col_1'].apply(type).value_counts()
Out[27]:
<class 'int'>    737858
<class 'str'>    262144
Name: col_1, dtype: int64

In [28]: mixed_df['col_1'].dtype
Out[28]: 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.

Specifying categorical dtype

New in version 0.19.0.

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

Scroll To Top

```
In [29]: data = ('col1,col2,col3\n'
   ....:         'a,b,1\n'
   ....:         'a,b,2\n'
   ....:         'c,d,3')
   ....:

In [30]: pd.read_csv(StringIO(data))
Out[30]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [31]: pd.read_csv(StringIO(data)).dtypes
Out[31]:
col1    object
col2    object
col3     int64
dtype: object

In [32]: pd.read_csv(StringIO(data), dtype='category').dtypes
Out[32]:
col1    category
col2    category
col3    category
dtype: object
```

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

```
In [33]: pd.read_csv(StringIO(data), dtype={'col1': 'category'}).dtypes
Out[33]:
col1    category
col2      object
col3       int64
dtype: object
```

New in version 0.21.0.

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 [34]: from pandas.api.types import CategoricalDtype

In [35]: dtype = CategoricalDtype(['d', 'c', 'b', 'a'], ordered=True)

In [36]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).dtypes
Out[36]:
col1    category
col2      object
col3       int64
dtype: object
```

Scroll To Top

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

```
In [37]: dtype = CategoricalDtype(['a', 'b', 'd'])  # No 'c'

In [38]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).col1
Out[38]:
0      a
1      a
2    NaN
Name: col1, dtype: category
Categories (3, object): [a, b, d]
```

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

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 [39]: df = pd.read_csv(StringIO(data), dtype='category')

In [40]: df.dtypes
Out[40]:
col1    category
col2    category
col3    category
dtype: object

In [41]: df['col3']
Out[41]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, object): [1, 2, 3]

In [42]: df['col3'].cat.categories = pd.to_numeric(df['col3'].cat.categories)

In [43]: df['col3']
Out[43]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, int64): [1, 2, 3]
```

Naming and using columns

Handling column names

Scroll To Top

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

```
In [44]: data = ('a,b,c\n'
   ....:         '1,2,3\n'
   ....:         '4,5,6\n'
   ....:         '7,8,9')
   ....:

In [45]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [46]: pd.read_csv(StringIO(data))
Out[46]:
   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 [47]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [48]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
Out[48]:
   foo  bar  baz
0    1    2    3
1    4    5    6
2    7    8    9

In [49]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)
Out[49]:
  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 [50]: data = ('skip this skip it\n'
   ....:         'a,b,c\n'
   ....:         '1,2,3\n'
   ....:         '4,5,6\n'
   ....:         '7,8,9')
   ....:

In [51]: pd.read_csv(StringIO(data), header=1)
Out[51]:
   a  b  c
0  1  2  3
```

Scroll To Top

```
1  4  5  6
2  7  8  9
```

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 [52]: data = ('a,b,a\n'
   ....:         '0,1,2\n'
   ....:         '3,4,5')
   ....:

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

There is no more duplicate data because `mangle_dupe_cols=True` by default, which modifies a series of duplicate columns ‘X’, …, ‘X’ to become ‘X’, ‘X.1’, …, ‘X.N’. If `mangle_dupe_cols=False`, duplicate data can arise:

```
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
Out[3]:
   a  b  a
0  2  1  2
1  5  4  5
```

To prevent users from encountering this problem with duplicate data, a `ValueError` exception is raised if `mangle_dupe_cols != True`:

```
In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
...
ValueError: Setting mangle_dupe_cols=False is not supported yet
```

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:

Scroll To Top

New in version 0.20.0: support for callable *usecols* arguments

```
In [54]: data = 'a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz'

In [55]: pd.read_csv(StringIO(data))
Out[55]:
   a  b  c    d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

In [56]: pd.read_csv(StringIO(data), usecols=['b', 'd'])
Out[56]:
   b    d
0  2  foo
1  5  bar
2  8  baz

In [57]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
Out[57]:
   a  c    d
0  1  3  foo
1  4  6  bar
2  7  9  baz

In [58]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['A', 'C'])
Out[58]:
   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 [59]: pd.read_csv(StringIO(data), usecols=lambda x: x not in ['a', 'c'])
Out[59]:
   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 [60]: data = ('\n'
   ....:         'a,b,c\n'
   ....:         '  \n'
   ....:         '# commented line\n'
   ....:         '1,2,3\n'
   ....:         '\n'
```

Scroll To Top

```
   ....:          '4,5,6')
   ....:

In [61]: print(data)

a,b,c

# commented line
1,2,3

4,5,6

In [62]: pd.read_csv(StringIO(data), comment='#')
Out[62]: 
   a  b  c
0  1  2  3
1  4  5  6
```

If `skip_blank_lines=False`, then `read_csv` will not ignore blank lines:

```
In [63]: data = ('a,b,c\n'
   ....:          '\n'
   ....:          '1,2,3\n'
   ....:          '\n'
   ....:          '\n'
   ....:          '4,5,6')
   ....:

In [64]: pd.read_csv(StringIO(data), skip_blank_lines=False)
Out[64]: 
     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 [65]: data = ('#comment\n'
   ....:          'a,b,c\n'
   ....:          'A,B,C\n'
   ....:          '1,2,3')
   ....:

In [66]: pd.read_csv(StringIO(data), comment='#', header=1)
Out[66]: 
   A  B  C
0  1  2  3

In [67]: data = ('A,B,C\n'
   ....:          '#comment\n'
   ....:          'a,b,c\n'
   ....:          '1,2,3')
   ....:

In [68]: pd.read_csv(StringIO(data), comment='#', skiprows=2)
```

Scroll To Top

```
Out[68]:
   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 [69]: 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 [70]: print(data)
# empty
# second empty line
# third emptyline
X,Y,Z
1,2,3
A,B,C
1,2.,4.
5.,NaN,10.0


In [71]: pd.read_csv(StringIO(data), comment='#', skiprows=4, header=1)
Out[71]:
     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 [72]: 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 [73]: df = pd.read_csv('tmp.csv')

In [74]: df
Out[74]:
         ID    level                        category
0  Patient1   123000           x # really unpleasant
1  Patient2    23000  y # wouldn't take his medicine
2  Patient3  1234018                     z # awesome
```

Scroll To Top

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

```
In [75]: df = pd.read_csv('tmp.csv', comment='#')

In [76]: df
Out[76]: 
         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 [77]: data = (b'word,length\n'
   ....:         b'Tr\xc3\xa4umen,7\n'
   ....:         b'Gr\xc3\xbc\xc3\x9fe,5')
   ....: 

In [78]: data = data.decode('utf8').encode('latin-1')

In [79]: df = pd.read_csv(BytesIO(data), encoding='latin-1')

In [80]: df
Out[80]: 
      word  length
0  Träumen       7
1    Grüße       5

In [81]: df['word'][1]
Out[81]: '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 [82]: data = ('a,b,c\n'
   ....:         '4,apple,bat,5.7\n'
   ....:         '8,orange,cow,10')
   ....: 

In [83]: pd.read_csv(StringIO(data))
Out[83]: 
        a    b     c
4   apple  bat   5.7
8  orange  cow  10.0
```

Scroll To Top

```
In [84]: data = ('index,a,b,c\n'
   ....:         '4,apple,bat,5.7\n'
   ....:         '8,orange,cow,10')
   ....:

In [85]: pd.read_csv(StringIO(data), index_col=0)
Out[85]:
            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 [86]: data = ('a,b,c\n'
   ....:         '4,apple,bat,\n'
   ....:         '8,orange,cow,')
   ....:

In [87]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [88]: pd.read_csv(StringIO(data))
Out[88]:
        a    b   c
4   apple  bat NaN
8  orange  cow NaN

In [89]: pd.read_csv(StringIO(data), index_col=False)
Out[89]:
   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 [90]: data = ('a,b,c\n'
   ....:         '4,apple,bat,\n'
   ....:         '8,orange,cow,')
   ....:

In [91]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [92]: pd.read_csv(StringIO(data), usecols=['b', 'c'])
Out[92]:
     b   c
4  bat NaN
8  cow NaN
```

Scroll To Top

```
In [93]: pd.read_csv(StringIO(data), usecols=['b', 'c'], index_col=0)
Out[93]:
     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_parser` 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`:

```
# Use a column as an index, and parse it as dates.
In [94]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)

In [95]: df
Out[95]:
            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 [96]: df.index
Out[96]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype='datetime64[ns
```

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 [97]: print(open('tmp.csv').read())
KORD,19990127, 19:00:00, 18:56:00, 0.8100
KORD,19990127, 20:00:00, 19:56:00, 0.0100
KORD,19990127, 21:00:00, 20:56:00, -0.5900
KORD,19990127, 21:00:00, 21:18:00, -0.9900
KORD,19990127, 22:00:00, 21:56:00, -0.5900
KORD,19990127, 23:00:00, 22:56:00, -0.5900

In [98]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])

In [99]: df
Out[99]:
                  1_2                 1_3     0     4
```

Scroll To Top

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

In [101]: df
Out[101]: 
                  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 [102]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [103]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)

In [104]: df
Out[104]: 
              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 [105]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [106]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   .....:                  index_col=0)  # index is the nominal column
   .....:
```

Scroll To Top

```
In [107]: df
Out[107]: 
                                  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.

Note: When passing a dict as the *parse_dates* argument, the order of the columns prepended is not guaranteed, because *dict* objects do not impose an ordering on their keys. On Python 2.7+ you may use *collections.OrderedDict* instead of a regular *dict* if this matters to you. Because of this, when using a dict for 'parse_dates' in conjunction with the *index_col* argument, it's best to specify *index_col* as a column label rather then as an index on the resulting frame.

Date parsing functions

Finally, the parser allows you to specify a custom `date_parser` function to take full advantage of the flexibility of the date parsing API:

```
In [108]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   .....:                  date_parser=pd.io.date_converters.parse_date_time)
   .....: 

In [109]: df
Out[109]: 
              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
```

Pandas will try to call the `date_parser` function in three different ways. If an exception is raised, the next one is tried:

1. `date_parser` is first called with one or more arrays as arguments, as defined using *parse_dates* (e.g., `date_parser(['2013', '2013'], ['1', '2'])`).
2. If #1 fails, `date_parser` is called with all the columns concatenated row-wise into a single array (e.g., `date_parser(['2013 1', '2013 2'])`).

Scroll To Top

3. If #2 fails, `date_parser` is called once for every row with one or more string arguments from the columns indicated with *parse_dates* (e.g., `date_parser('2013', '1')` for the first row, `date_parser('2013', '2')` for the second, etc.).

Note that performance-wise, you should try these methods of parsing dates in order:

1. Try to infer the format using `infer_datetime_format=True` (see section below).
2. If you know the format, use `pd.to_datetime()`: `date_parser=lambda x: pd.to_datetime(x, format=...)`.
3. If you have a really non-standard format, use a custom `date_parser` function. For optimal performance, this should be vectorized, i.e., it should accept arrays as arguments.

You can explore the date parsing functionality in date_converters.py and add your own. We would love to turn this module into a community supported set of date/time parsers. To get you started, `date_converters.py` contains functions to parse dual date and time columns, year/month/day columns, and year/month/day/hour/minute/second columns. It also contains a `generic_parser` function so you can curry it with a function that deals with a single date rather than the entire array.

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`.

```
In [110]: content = """\
   .....: a
   .....: 2000-01-01T00:00:00+05:00
   .....: 2000-01-01T00:00:00+06:00"""
   .....:

In [111]: df = pd.read_csv(StringIO(content), parse_dates=['a'])

In [112]: df['a']
Out[112]:
0    2000-01-01 00:00:00+05:00
1    2000-01-01 00:00:00+06:00
Name: a, dtype: object
```

To parse the mixed-timezone values as a datetime column, pass a partially-applied **`to_datetime()`** with `utc=True` as the `date_parser`.

```
In [113]: df = pd.read_csv(StringIO(content), parse_dates=['a'],
   .....:                  date_parser=lambda col: pd.to_datetime(col, utc=True))
   .....:

In [114]: df['a']
Out[114]:
0   1999-12-31 19:00:00+00:00
1   1999-12-31 18:00:00+00:00
Name: a, dtype: datetime64[ns, UTC]
```

Scroll To Top

Inferring datetime format

If you have `parse_dates` enabled for some or all of your columns, and your datetime strings are all formatted the same way, you may get a large speed up by setting `infer_datetime_format=True`. If set, pandas will attempt to guess the format of your datetime strings, and then use a faster means of parsing the strings. 5-10x parsing speeds have been observed. pandas will fallback to the usual parsing if either the format cannot be guessed or the format that was guessed cannot properly parse the entire column of strings. So in general, `infer_datetime_format` should not have any negative consequences if enabled.

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

- "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 `infer_datetime_format` 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.

```
# Try to infer the format for the index column
In [115]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
   .....:                  infer_datetime_format=True)
   .....:

In [116]: df
Out[116]:
            A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5
```

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 [117]: print(open('tmp.csv').read())
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c

In [118]: pd.read_csv('tmp.csv', parse_dates=[0])
Out[118]:
        date  value cat
0 2000-01-06      5   a
1 2000-02-06     10   b
2 2000-03-06     15   c
```

Scroll To Top

```
In [119]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])
Out[119]:
        date  value cat
0 2000-06-01      5   a
1 2000-06-02     10   b
2 2000-06-03     15   c
```

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 [120]: val = '0.3066101993807095471566981359501369297504425048828125'

In [121]: data = 'a,b,c\n1,2,{0}'.format(val)

In [122]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                 float_precision=None)['c'][0] - float(val))
   .....:
Out[122]: 1.1102230246251565e-16

In [123]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                 float_precision='high')['c'][0] - float(val))
   .....:
Out[123]: 5.551115123125783e-17

In [124]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                 float_precision='round_trip')['c'][0] - float(val))
   .....:
Out[124]: 0.0
```

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 [125]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [126]: df = pd.read_csv('tmp.csv', sep='|')

In [127]: df
Out[127]:
         ID      level category
0  Patient1    123,000        x
1  Patient2     23,000        y
2  Patient3  1,234,018        z
```

Scroll To Top

```
In [128]: df.level.dtype
Out[128]: dtype('O')
```

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

```
In [129]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [130]: df = pd.read_csv('tmp.csv', sep='|', thousands=',')

In [131]: df
Out[131]: 
         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z

In [132]: df.level.dtype
Out[132]: dtype('int64')
```

NA values

To control which values are parsed as missing values (which are signified by `NaN`), specify a string in `na_values`. If you specify a list of strings, then all values in it are considered to be missing 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', 'NULL', 'null', 'NaN', '-NaN', 'nan', '-nan', '']`.

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`.

Scroll To Top

```
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`.

Returning Series

Using the `squeeze` keyword, the parser will return output with a single column as a `Series`:

```
In [133]: print(open('tmp.csv').read())
level
Patient1,123000
Patient2,23000
Patient3,1234018

In [134]: output = pd.read_csv('tmp.csv', squeeze=True)

In [135]: output
Out[135]:
Patient1     123000
Patient2      23000
Patient3    1234018
Name: level, dtype: int64

In [136]: type(output)
Out[136]: pandas.core.series.Series
```

Boolean values

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 [137]: data = ('a,b,c\n'
   .....:         '1,Yes,2\n'
   .....:         '3,No,4')
   .....:

In [138]: print(data)
a,b,c
1,Yes,2
3,No,4

In [139]: pd.read_csv(StringIO(data))
```

Scroll To Top

```
Out[139]:
   a    b  c
0  1  Yes  2
1  3   No  4

In [140]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
Out[140]:
   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 [141]: data = ('a,b,c\n'
   .....:         '1,2,3\n'
   .....:         '4,5,6,7\n'
   .....:         '8,9,10')
   .....:

In [142]: pd.read_csv(StringIO(data))
---------------------------------------------------------------------------
ParserError                               Traceback (most recent call last)
<ipython-input-142-6388c394e6b8> in <module>
----> 1 pd.read_csv(StringIO(data))

/pandas/pandas/io/parsers.py in parser_f(filepath_or_buffer, sep, delimiter, header, na
    683         )
    684
--> 685         return _read(filepath_or_buffer, kwds)
    686
    687     parser_f.__name__ = name

/pandas/pandas/io/parsers.py in _read(filepath_or_buffer, kwds)
    461
    462     try:
--> 463         data = parser.read(nrows)
    464     finally:
    465         parser.close()

/pandas/pandas/io/parsers.py in read(self, nrows)
   1152     def read(self, nrows=None):
   1153         nrows = _validate_integer("nrows", nrows)
-> 1154         ret = self._engine.read(nrows)
   1155
   1156         # May alter columns / col_dict

/pandas/pandas/io/parsers.py in read(self, nrows)
   2046     def read(self, nrows=None):
   2047         try:
-> 2048             data = self._reader.read(nrows)
   2049         except StopIteration:
   2050             if self._first_chunk:

/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader.read()

/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._read_low_memory()
```

Scroll To Top

```
/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._read_rows()

/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._tokenize_rows()

/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.raise_parser_error()

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

You can elect to skip bad lines:

```
In [29]: pd.read_csv(StringIO(data), error_bad_lines=False)
Skipping line 3: expected 3 fields, saw 4

Out[29]:
   a  b   c
0  1  2   3
1  8  9  10
```

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

```
In [30]: pd.read_csv(StringIO(data), usecols=[0, 1, 2])

 Out[30]:
    a  b   c
 0  1  2   3
 1  4  5   6
 2  8  9  10
```

Dialect

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 [143]: 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 [144]: import csv

In [145]: dia = csv.excel()

In [146]: dia.quoting = csv.QUOTE_NONE
```

Scroll To Top

```
In [147]: pd.read_csv(StringIO(data), dialect=dia)
Out[147]:
       label1 label2 label3
index1     "a      c      e
index2      b      d      f
```

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

```
In [148]: data = 'a,b,c~1,2,3~4,5,6'

In [149]: pd.read_csv(StringIO(data), lineterminator='~')
Out[149]:
   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 [150]: data = 'a, b, c\n1, 2, 3\n4, 5, 6'

In [151]: print(data)
a, b, c
1, 2, 3
4, 5, 6

In [152]: pd.read_csv(StringIO(data), skipinitialspace=True)
Out[152]:
   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 [153]: data = 'a,b\n"hello, \\"Bob\\", nice to see you",5'

In [154]: print(data)
a,b
"hello, \"Bob\", nice to see you",5

In [155]: pd.read_csv(StringIO(data), escapechar='\\')
Out[155]:
                               a  b
0  hello, "Bob", nice to see you  5
```

Scroll To Top

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 specify the filler character of the fields if it is not spaces (e.g., '~').

Consider a typical fixed-width data file:

```
In [156]: print(open('bar.csv').read())
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
```

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 [157]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [158]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)

In [159]: df
Out[159]: 
                 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 [160]: widths = [6, 14, 13, 10]

In [161]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [162]: df
Out[162]: 
        0           1           2        3
0  id8141  360.242940  149.910199  11950.7
1  id1594  444.953632  166.985655  11788.4
```

Scroll To Top

```
2  id1849  364.136849  183.628767  11806.2
3  id1230  413.836124  184.375703  11916.8
4  id1948  502.953953  173.237159  12468.3
```

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 [163]: df = pd.read_fwf('bar.csv', header=None, index_col=0)

In [164]: df
Out[164]: 
                 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
```

New in version 0.20.0.

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

```
In [165]: pd.read_fwf('bar.csv', header=None, index_col=0).dtypes
Out[165]: 
1    float64
2    float64
3    float64
dtype: object

In [166]: pd.read_fwf('bar.csv', header=None, dtype={2: 'object'}).dtypes
Out[166]:
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:

Scroll To Top

```
In [167]: print(open('foo.csv').read())
A,B,C
```

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

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

```
In [168]: pd.read_csv('foo.csv')
Out[168]:
          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 [169]: df = pd.read_csv('foo.csv', parse_dates=True)

In [170]: df.index
Out[170]: 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 [171]: print(open('data/mindex_ex.csv').read())
year,indiv,zit,xit
1977,"A",1.2,.6
1977,"B",1.5,.5
1977,"C",1.7,.8
1978,"A",.2,.06
1978,"B",.7,.2
1978,"C",.8,.3
1978,"D",.9,.5
1978,"E",1.4,.9
1979,"C",.2,.15
1979,"D",.14,.05
1979,"E",.5,.15
1979,"F",1.2,.5
1979,"G",3.4,1.9
1979,"H",5.4,2.7
1979,"I",6.4,1.2
```

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:

```
In [172]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0, 1])

In [173]: df
Out[173]:
             zit   xit
year indiv
1977 A      1.20  0.60
     B      1.50  0.50
```

Scroll To Top

```
     C       1.70  0.80
1978 A       0.20  0.06
     B       0.70  0.20
     C       0.80  0.30
     D       0.90  0.50
     E       1.40  0.90
1979 C       0.20  0.15
     D       0.14  0.05
     E       0.50  0.15
     F       1.20  0.50
     G       3.40  1.90
     H       5.40  2.70
     I       6.40  1.20

In [174]: df.loc[1978]
Out[174]:
       zit   xit
indiv
A      0.2  0.06
B      0.7  0.20
C      0.8  0.30
D      0.9  0.50
E      1.4  0.90
```

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 [175]: from pandas.util.testing import makeCustomDataframe as mkdf

In [176]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)

In [177]: df.to_csv('mi.csv')

In [178]: print(open('mi.csv').read())
C0,,C_l0_g0,C_l0_g1,C_l0_g2
C1,,C_l1_g0,C_l1_g1,C_l1_g2
C2,,C_l2_g0,C_l2_g1,C_l2_g2
C3,,C_l3_g0,C_l3_g1,C_l3_g2
R0,R1,,,
R_l0_g0,R_l1_g0,R0C0,R0C1,R0C2
R_l0_g1,R_l1_g1,R1C0,R1C1,R1C2
R_l0_g2,R_l1_g2,R2C0,R2C1,R2C2
R_l0_g3,R_l1_g3,R3C0,R3C1,R3C2
R_l0_g4,R_l1_g4,R4C0,R4C1,R4C2


In [179]: pd.read_csv('mi.csv', header=[0, 1, 2, 3], index_col=[0, 1])
Out[179]:
C0                C_l0_g0 C_l0_g1 C_l0_g2
C1                C_l1_g0 C_l1_g1 C_l1_g2
C2                C_l2_g0 C_l2_g1 C_l2_g2
C3                C_l3_g0 C_l3_g1 C_l3_g2
R0      R1
R_l0_g0 R_l1_g0      R0C0    R0C1    R0C2
R_l0_g1 R_l1_g1      R1C0    R1C1    R1C2
R_l0_g2 R_l1_g2      R2C0    R2C1    R2C2
R_l0_g3 R_l1_g3      R3C0    R3C1    R3C2
R_l0_g4 R_l1_g4      R4C0    R4C1    R4C2
```

Scroll To Top

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

```
In [180]: print(open('mi2.csv').read())
,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 [181]: pd.read_csv('mi2.csv', header=[0, 1], index_col=0)
Out[181]:
     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 [182]: print(open('tmp2.sv').read())
:0:1:2:3
0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.1356323710171934
1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-1.0442359662799567
2:-0.8618489633477999:-2.1045692188948086:-0.4949292740687813:1.071803807037338
3:0.7215551622443669:-0.7067711336300845:-1.0395749851146963:0.27185988554282986
4:-0.42497232978883753:0.567020349793672:0.27623201927771873:-1.0874006912859915
5:-0.6736897080883706:0.113648409688855:-1.4784265524372235:0.5249876671147047
6:0.4047052186802365:0.5770459859204836:-1.7150020161146375:-1.0392684835147725
7:-0.3706468582364464:-1.1578922506419993:-1.344311812731667:0.8448851414248841
8:1.0757697837155533:-0.10904997528022223:1.6435630703622064:-1.4693879595399115
9:0.35702056413309086:-0.6746001037299882:-1.776903716971867:-0.9689138124473498


In [183]: pd.read_csv('tmp2.sv', sep=None, engine='python')
Out[183]:
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
8           8  1.075770 -0.109050  1.643563 -1.469388
9           9  0.357021 -0.674600 -1.776904 -0.968914
```

Reading multiple files to create a single DataFrame

Scroll To Top

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 [184]: print(open('tmp.sv').read())
|0|1|2|3
0|0.4691122999071863|-0.2828633443286633|-1.5090585031735124|-1.1356323710171934
1|1.2121120250208506|-0.17321464905330858|0.11920871129693428|-1.0442359662799567
2|-0.8618489633477999|-2.1045692188948086|-0.4949292740687813|1.071803807037338
3|0.7215551622443669|-0.7067711336300845|-1.0395749851146963|0.27185988554282986
4|-0.42497232978883753|0.567020349793672|0.27623201927771873|-1.0874006912859915
5|-0.6736897080883706|0.113648409688855|-1.4784265524372235|0.5249876671147047
6|0.4047052186802365|0.5770459859204836|-1.7150020161146375|-1.0392684835147725
7|-0.3706468582364464|-1.1578922506419993|-1.344311812731667|0.8448851414248841
8|1.0757697837155533|-0.10904997528022223|1.6435630703622064|-1.4693879595399115
9|0.35702056413309086|-0.6746001037299882|-1.776903716971867|-0.9689138124473498


In [185]: table = pd.read_csv('tmp.sv', sep='|')

In [186]: table
Out[186]:
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
8           8  1.075770 -0.109050  1.643563 -1.469388
9           9  0.357021 -0.674600 -1.776904 -0.968914
```

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

```
In [187]: reader = pd.read_csv('tmp.sv', sep='|', chunksize=4)

In [188]: reader
Out[188]: <pandas.io.parsers.TextFileReader at 0x7f65f17cf7f0>

In [189]: for chunk in reader:
   .....:     print(chunk)
   .....:
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
   Unnamed: 0         0         1         2         3
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
   Unnamed: 0         0        1         2         3
8           8  1.075770 -0.10905  1.643563 -1.469388
9           9  0.357021 -0.67460 -1.776904 -0.968914
```

Scroll To Top

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

```
In [190]: reader = pd.read_csv('tmp.sv', sep='|', iterator=True)

In [191]: reader.get_chunk(5)
Out[191]: 
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
```

Specifying the parser engine

Under the hood pandas uses a fast and efficient parser implemented in C as well as a Python implementation which is currently more feature-complete. Where possible pandas uses the C parser (specified as `engine='c'`), but may fall back to Python if C-unsupported options are specified. Currently, C-unsupported options 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'`.

Reading remote files

You can pass in a URL to a CSV file:

```
df = pd.read_csv('https://download.bls.gov/pub/time.series/cu/cu.item',
                 sep='\t')
```

S3 URLs are handled as well but require installing the S3Fs library:

```
df = pd.read_csv('s3://pandas-test/tips.csv')
```

If your S3 bucket requires credentials you will need to set them as environment variables or in the `~/.aws/credentials` config file, refer to the S3Fs documentation on credentials.

Writing out data

Scroll To Top

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
- `line_terminator`: Character sequence denoting line end (default *os.linesep*)
- `quoting`: Set quoting rules as in csv module (default csv.QUOTE_MINIMAL). Note that if you have set a *float_format* then floats are converted to strings and csv.QUOTE_NONNUMERIC 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)

Scroll To Top

- `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.

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

- `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.
- `lines` : If `records` orient, then will write each record per line as json.

Scroll To Top

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

In [193]: json = dfj.to_json()

In [194]: json
Out[194]: '{"A":{"0":-1.2945235903,"1":0.2766617129,"2":-0.0139597524,"3":-0.0061535699
```

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

In [196]: dfjo
Out[196]:
   A  B  C
x  1  4  7
y  2  5  8
z  3  6  9

In [197]: sjo = pd.Series(dict(x=15, y=16, z=17), name='D')

In [198]: sjo
Out[198]:
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 [199]: dfjo.to_json(orient="columns")
Out[199]: '{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":{"x":7,"y":8,"z":9}}'

# Not available for Series
```

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

```
In [200]: dfjo.to_json(orient="index")
Out[200]: '{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":{"A":3,"B":6,"C":9}}'

In [201]: sjo.to_json(orient="index")
Out[201]: '{"x":15,"y":16,"z":17}'
```

Scroll To Top

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

In [203]: sjo.to_json(orient="records")
Out[203]: '[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 [204]: dfjo.to_json(orient="values")
Out[204]: '[[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 [205]: dfjo.to_json(orient="split")
Out[205]: '{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,5,8],[3,6,

In [206]: sjo.to_json(orient="split")
Out[206]: '{"name":"D","index":["x","y","z"],"data":[15,16,17]}'
```

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

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

Date handling

Writing in ISO date format:

```
In [207]: dfd = pd.DataFrame(np.random.randn(5, 2), columns=list('AB'))

In [208]: dfd['date'] = pd.Timestamp('20130101')

In [209]: dfd = dfd.sort_index(1, ascending=False)

In [210]: json = dfd.to_json(date_format='iso')

In [211]: json
Out[211]: '{"date":{"0":"2013-01-01T00:00:00.000Z","1":"2013-01-01T00:00:00.000Z","2":"
```

Scroll To Top

Writing in ISO date format, with microseconds:

```
In [212]: json = dfd.to_json(date_format='iso', date_unit='us')

In [213]: json
Out[213]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00:00.000000Z"
```

Epoch timestamps, in seconds:

```
In [214]: json = dfd.to_json(date_format='epoch', date_unit='s')

In [215]: json
Out[215]: '{"date":{"0":1356998400,"1":1356998400,"2":1356998400,"3":1356998400,"4":135
```

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

```
In [216]: dfj2 = dfj.copy()

In [217]: dfj2['date'] = pd.Timestamp('20130101')

In [218]: dfj2['ints'] = list(range(5))

In [219]: dfj2['bools'] = True

In [220]: dfj2.index = pd.date_range('20130101', periods=5)

In [221]: dfj2.to_json('test.json')

In [222]: with open('test.json') as fh:
   .....:     print(fh.read())
   .....:
{"A":{"1356998400000":-1.2945235903,"1357084800000":0.2766617129,"1357171200000":-0.013
```

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:

Scroll To Top

```
>>> 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 [223]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
Out[223]: '{"0":{"0":"(1+0j)","1":"(2+0j)","2":"(1+2j)"}}'
```

Reading JSON

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

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

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

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

  The format of the JSON string

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

- `dtype` : if True, infer dtypes, if a dict of column to dtype, then use those, if `False`, then don't infer dtypes at all, default is True, apply only to the data.
- `convert_axes` : boolean, try to convert the axes to the proper dtypes, default is `True`
- `convert_dates` : a list of columns to parse for dates; If `True`, then try to parse date-like columns, default is `True`.
- `keep_default_dates` : boolean, default `True`. If parsing dates, then parse the default date-like columns.

Scroll To Top

- `numpy` : direct decoding to NumPy arrays. default is `False`; Supports numeric data only, although labels may be non-numeric. Also note that the JSON ordering **MUST** be the same for each term if `numpy=True`.
- `precise_float` : boolean, default `False`. Set to enable usage of higher precision (strtod) function when decoding string to double values. Default (`False`) is to use fast but less precise builtin functionality.
- `date_unit` : string, the timestamp unit to detect if converting dates. Default None. By default the timestamp precision will be detected, if this is not desired then pass one of 's', 'ms', 'us' or 'ns' to force timestamp precision to seconds, milliseconds, microseconds or nanoseconds respectively.
- `lines` : reads file as one json object per line.
- `encoding` : The encoding to use to decode py3 bytes.
- `chunksize` : when used in combination with `lines=True`, return a JsonReader which reads in `chunksize` lines per iteration.

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

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

Data conversion

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

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

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

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

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

Scroll To Top

- 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 [224]: pd.read_json(json)
Out[224]:
        date         B         A
0 2013-01-01  2.565646 -1.206412
1 2013-01-01  1.340309  1.431256
2 2013-01-01 -0.226169 -1.170299
3 2013-01-01  0.813850  0.410835
4 2013-01-01 -0.827317  0.132003
```

Reading from a file:

```
In [225]: pd.read_json('test.json')
Out[225]:
                   A         B       date  ints  bools
2013-01-01 -1.294524  0.413738 2013-01-01     0   True
2013-01-02  0.276662 -0.472035 2013-01-01     1   True
2013-01-03 -0.013960 -0.362543 2013-01-01     2   True
2013-01-04 -0.006154 -0.923061 2013-01-01     3   True
2013-01-05  0.895717  0.805244 2013-01-01     4   True
```

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

```
In [226]: pd.read_json('test.json', dtype=object).dtypes
Out[226]:
A        object
B        object
date     object
ints     object
bools    object
dtype: object
```

Specify dtypes for conversion:

```
In [227]: pd.read_json('test.json', dtype={'A': 'float32', 'bools': 'int8'}).dtypes
Out[227]:
A                 float32
B                 float64
date       datetime64[ns]
ints                int64
bools                int8
dtype: object
```

Preserve string indices:

Scroll To Top

```
In [228]: si = pd.DataFrame(np.zeros((4, 4)), columns=list(range(4)),
   .....:                   index=[str(i) for i in range(4)])
   .....:

In [229]: si
Out[229]:
     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 [230]: si.index
Out[230]: Index(['0', '1', '2', '3'], dtype='object')

In [231]: si.columns
Out[231]: Int64Index([0, 1, 2, 3], dtype='int64')

In [232]: json = si.to_json()

In [233]: sij = pd.read_json(json, convert_axes=False)

In [234]: sij
Out[234]:
   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 [235]: sij.index
Out[235]: Index(['0', '1', '2', '3'], dtype='object')

In [236]: sij.columns
Out[236]: Index(['0', '1', '2', '3'], dtype='object')
```

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

```
In [237]: json = dfj2.to_json(date_unit='ns')

# Try to parse timestamps as milliseconds -> Won't Work
In [238]: dfju = pd.read_json(json, date_unit='ms')

In [239]: dfju
Out[239]:
                            A         B                 date  ints  bools
1356998400000000000 -1.294524  0.413738  1356998400000000000     0   True
1357084800000000000  0.276662 -0.472035  1356998400000000000     1   True
1357171200000000000 -0.013960 -0.362543  1356998400000000000     2   True
1357257600000000000 -0.006154 -0.923061  1356998400000000000     3   True
1357344000000000000  0.895717  0.805244  1356998400000000000     4   True

# Let pandas detect the correct precision
In [240]: dfju = pd.read_json(json)

In [241]: dfju
Out[241]:
                   A         B       date  ints  bools
2013-01-01 -1.294524  0.413738 2013-01-01     0   True
2013-01-02  0.276662 -0.472035 2013-01-01     1   True
2013-01-03 -0.013960 -0.362543 2013-01-01     2   True
2013-01-04 -0.006154 -0.923061 2013-01-01     3   True
```

Scroll To Top

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

# Or specify that all timestamps are in nanoseconds
In [242]: dfju = pd.read_json(json, date_unit='ns')

In [243]: dfju
Out[243]: 
                   A         B       date  ints  bools
2013-01-01 -1.294524  0.413738 2013-01-01     0   True
2013-01-02  0.276662 -0.472035 2013-01-01     1   True
2013-01-03 -0.013960 -0.362543 2013-01-01     2   True
2013-01-04 -0.006154 -0.923061 2013-01-01     3   True
2013-01-05  0.895717  0.805244 2013-01-01     4   True
```

The Numpy parameter

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

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

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

```
In [244]: randfloats = np.random.uniform(-100, 1000, 10000)

In [245]: randfloats.shape = (1000, 10)

In [246]: dffloats = pd.DataFrame(randfloats, columns=list('ABCDEFGHIJ'))

In [247]: jsonfloats = dffloats.to_json()
```

```
In [248]: %timeit pd.read_json(jsonfloats)
12.4 ms +- 116 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [249]: %timeit pd.read_json(jsonfloats, numpy=True)
9.56 ms +- 82.8 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

The speedup is less noticeable for smaller datasets:

```
In [250]: jsonfloats = dffloats.head(100).to_json()
```

```
In [251]: %timeit pd.read_json(jsonfloats)
8.05 ms +- 120 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Scroll To Top

```
In [252]: %timeit pd.read_json(jsonfloats, numpy=True)
7 ms +- 162 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

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

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

Normalization

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

```
In [253]: from pandas.io.json import json_normalize

In [254]: data = [{'id': 1, 'name': {'first': 'Coleen', 'last': 'Volk'}},
   .....:         {'name': {'given': 'Mose', 'family': 'Regner'}},
   .....:         {'id': 2, 'name': 'Faye Raker'}]
   .....:

In [255]: json_normalize(data)
Out[255]:
    id name.first name.last name.given name.family        name
0  1.0     Coleen      Volk        NaN         NaN         NaN
1  NaN        NaN       NaN       Mose      Regner         NaN
2  2.0        NaN       NaN        NaN         NaN  Faye Raker
```

```
In [256]: data = [{'state': 'Florida',
   .....:          'shortname': 'FL',
   .....:          'info': {'governor': 'Rick Scott'},
   .....:          'counties': [{'name': 'Dade', 'population': 12345},
   .....:                       {'name': 'Broward', 'population': 40000},
   .....:                       {'name': 'Palm Beach', 'population': 60000}]},
   .....:         {'state': 'Ohio',
   .....:          'shortname': 'OH',
   .....:          'info': {'governor': 'John Kasich'},
   .....:          'counties': [{'name': 'Summit', 'population': 1234},
   .....:                       {'name': 'Cuyahoga', 'population': 1337}]}]
   .....:

In [257]: json_normalize(data, 'counties', ['state', 'shortname', ['info', 'governor']]
Out[257]:
         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
```

Scroll To Top

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

In [259]: json_normalize(data, max_level=1)
Out[259]:
  CreatedBy.Name Lookup.TextField                    Lookup.UserField Image.a
0        User001        Some text  {'Id': 'ID001', 'Name': 'Name001'}       b
```

Line delimited json

New in version 0.19.0.

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

New in version 0.21.0.

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

```
In [260]: jsonl = '''
   .....:     {"a": 1, "b": 2}
   .....:     {"a": 3, "b": 4}
   .....: '''
   .....:

In [261]: df = pd.read_json(jsonl, lines=True)

In [262]: df
Out[262]:
   a  b
0  1  2
1  3  4

In [263]: df.to_json(orient='records', lines=True)
Out[263]: '{"a":1,"b":2}\n{"a":3,"b":4}'

# reader is an iterator that returns `chunksize` lines each iteration
In [264]: reader = pd.read_json(StringIO(jsonl), lines=True, chunksize=1)

In [265]: reader
Out[265]: <pandas.io.json._json.JsonReader at 0x7f65f15bac18>

In [266]: for chunk in reader:
   .....:     print(chunk)
   .....:
Empty DataFrame
Columns: []
Index: []
```

Scroll To Top

```
   a  b
0  1  2
   a  b
1  3  4
```

Table schema

New in version 0.20.0.

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 [267]: 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 [268]: df
Out[268]: 
     A  B          C
idx                 
0    1  a 2016-01-01
1    2  b 2016-01-02
2    3  c 2016-01-03

In [269]: df.to_json(orient='table', date_format="iso")
Out[269]: '{"schema": {"fields":[{"name":"idx","type":"integer"},{"name":"A","type":"in
```

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

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

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

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

Scroll To Top

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 [270]: from pandas.io.json import build_table_schema

In [271]: s = pd.Series(pd.date_range('2016', periods=4))

In [272]: build_table_schema(s)
Out[272]: 
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'datetime'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

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

```
In [273]: s_tz = pd.Series(pd.date_range('2016', periods=12,
   .....:                                tz='US/Central'))
   .....: 

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

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

```
In [275]: s_per = pd.Series(1, index=pd.period_range('2016', freq='A-DEC',
   .....:                                            periods=4))
   .....: 

In [276]: build_table_schema(s_per)
Out[276]: 
{'fields': [{'name': 'index', 'type': 'datetime', 'freq': 'A-DEC'},
  {'name': 'values', 'type': 'integer'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

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

Scroll To Top

```
In [277]: s_cat = pd.Series(pd.Categorical(['a', 'b', 'a']))

In [278]: build_table_schema(s_cat)
Out[278]: 
```

```
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values',
   'type': 'any',
   'constraints': {'enum': ['a', 'b']},
   'ordered': False}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

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

```
In [279]: s_dupe = pd.Series([1, 2], index=[1, 1])

In [280]: build_table_schema(s_dupe)
Out[280]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
 'pandas_version': '0.20.0'}
```

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

```
In [281]: s_multi = pd.Series(1, index=pd.MultiIndex.from_product([('a', 'b'),
   .....:                                                          (0, 1)]))
   .....:

In [282]: build_table_schema(s_multi)
Out[282]:
{'fields': [{'name': 'level_0', 'type': 'string'},
  {'name': 'level_1', 'type': 'integer'},
  {'name': 'values', 'type': 'integer'}],
 'primaryKey': FrozenList(['level_0', 'level_1']),
 'pandas_version': '0.20.0'}
```

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

New in version 0.23.0.

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

```
In [283]: 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 [284]: df
```

Scroll To Top

```
Out[284]:
     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 [285]: df.dtypes
Out[285]:
foo             int64
bar            object
baz    datetime64[ns]
qux          category
dtype: object

In [286]: df.to_json('test.json', orient='table')

In [287]: new_df = pd.read_json('test.json', orient='table')

In [288]: new_df
Out[288]:
     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 [289]: new_df.dtypes
Out[289]:
foo             int64
bar            object
baz    datetime64[ns]
qux          category
dtype: object
```

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

```
In [290]: df.index.name = 'index'

In [291]: df.to_json('test.json', orient='table')

In [292]: new_df = pd.read_json('test.json', orient='table')

In [293]: print(new_df.index.name)
None
```

HTML

Reading HTML content

Scroll To Top

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

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

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

Read a URL with no options:

```
In [294]: url = 'https://www.fdic.gov/bank/individual/failed/banklist.html'

In [295]: dfs = pd.read_html(url)

In [296]: dfs
Out[296]: 
[                                             Bank Name        City  ST   CERT  
 0                                 The Enloe State Bank      Cooper  TX  10716   
 1                  Washington Federal Bank for Savings     Chicago  IL  30570   
 2      The Farmers and Merchants State Bank of Argonia     Argonia  KS  17719   
 3                                  Fayette County Bank  Saint Elmo  IL   1802   
 4    Guaranty Bank, (d/b/a BestBank in Georgia & Mi...   Milwaukee  WI  30003  First-C
 ..                                                 ...         ...  ..    ...   
 551                                Superior Bank, FSB    Hinsdale  IL  32646   
 552                               Malta National Bank       Malta  OH   6629   
 553                   First Alliance Bank & Trust Co.  Manchester  NH  34264  Souther
 554                 National State Bank of Metropolis  Metropolis  IL   3815   
 555                                  Bank of Honolulu    Honolulu  HI  21029   

 [556 rows x 7 columns]]
```

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

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

```
In [297]: with open(file_path, 'r') as f:
   .....:     dfs = pd.read_html(f.read())
   .....: 

In [298]: dfs
Out[298]: 
[                                   Bank Name          City  ST   CERT  
 0    Banks of Wisconsin d/b/a Bank of Kenosha       Kenosha  WI  35386   
 1                        Central Arizona Bank    Scottsdale  AZ  34527   
 2                                Sunrise Bank      Valdosta  GA  58185   
 3                       Pisgah Community Bank     Asheville  NC  58701   
 4                         Douglas County Bank  Douglasville  GA  21649   
 ..                                        ...           ...  ..    ...   
 500                        Superior Bank, FSB      Hinsdale  IL  32646   
 501                       Malta National Bank         Malta  OH   6629   
 502           First Alliance Bank & Trust Co.    Manchester  NH  34264  Southern New H
 503         National State Bank of Metropolis    Metropolis  IL   3815              Ba
 504                          Bank of Honolulu      Honolulu  HI  21029   
```

Scroll To Top

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

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

```
In [299]: with open(file_path, 'r') as f:
   .....:     sio = StringIO(f.read())
   .....:

In [300]: dfs = pd.read_html(sio)

In [301]: dfs
Out[301]:
[                             Bank Name          City  ST   CERT
 0    Banks of Wisconsin d/b/a Bank of Kenosha       Kenosha  WI  35386
 1                Central Arizona Bank    Scottsdale  AZ  34527
 2                        Sunrise Bank      Valdosta  GA  58185
 3               Pisgah Community Bank     Asheville  NC  58701
 4                 Douglas County Bank  Douglasville  GA  21649
 ..                                ...           ...  ..    ...
 500                Superior Bank, FSB      Hinsdale  IL  32646
 501               Malta National Bank         Malta  OH   6629
 502         First Alliance Bank & Trust Co.    Manchester  NH  34264  Southern New H
 503       National State Bank of Metropolis    Metropolis  IL   3815            Ba
 504                  Bank of Honolulu      Honolulu  HI  21029

 [505 rows x 7 columns]]
```

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

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

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

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

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

Specify an index column:

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

Scroll To Top

Specify a number of rows to skip:

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

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

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

Specify an HTML attribute:

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

Specify values that should be converted to NaN:

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

New in version 0.19.

Specify whether to keep the default set of NaN values:

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

New in version 0.19.

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

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

New in version 0.19.

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)
```

Scroll To Top

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

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

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

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

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

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

Writing to HTML files

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

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

```
In [302]: df = pd.DataFrame(np.random.randn(2, 2))

In [303]: df
Out[303]: 
          0         1
0 -0.184744  0.496971
1 -0.856240  1.857977

In [304]: print(df.to_html())  # raw html
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
```

Scroll To Top

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

HTML:

	0	1
0	-0.184744	0.496971
1	-0.856240	1.857977

The `columns` argument will limit the columns shown:

```
In [305]: print(df.to_html(columns=[0]))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
    </tr>
  </tbody>
</table>
```

HTML:

	0
0	-0.184744
1	-0.856240

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

```
In [306]: print(df.to_html(float_format='{0:.10f}'.format))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.1847438576</td>
```

Scroll To Top

```
      <td>0.4969711327</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.8562396763</td>
      <td>1.8579766508</td>
    </tr>
  </tbody>
</table>
```

HTML:

	0	1
0	-0.1847438576	0.4969711327
1	-0.8562396763	1.8579766508

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

```
In [307]: print(df.to_html(bold_rows=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>0</td>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <td>1</td>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

	0	1
0	-0.184744	0.496971
1	-0.856240	1.857977

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.

Scroll To Top

```
In [308]: print(df.to_html(classes=['awesome_table_class', 'even_more_awesome_class']))
<table border="1" class="dataframe awesome_table_class even_more_awesome_class">
  <thead>
    <tr style="text-align: right;">
```

```
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

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

New in version 0.24.

```
In [309]: url_df = pd.DataFrame({
   .....:     'name': ['Python', 'Pandas'],
   .....:     'url': ['https://www.python.org/', 'http://pandas.pydata.org']})
   .....:

In [310]: print(url_df.to_html(render_links=True))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>name</th>
      <th>url</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>Python</td>
      <td><a href="https://www.python.org/" target="_blank">https://www.python.org/</a>
    </tr>
    <tr>
      <th>1</th>
      <td>Pandas</td>
      <td><a href="http://pandas.pydata.org" target="_blank">http://pandas.pydata.org</
    </tr>
  </tbody>
</table>
```

HTML:

	name	url
0	Python	https://www.python.org/
1	Pandas	http://pandas.pydata.org

Scroll To Top

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

Escaped:

```
In [312]: print(df.to_html())
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>&amp;</td>
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>
      <td>&lt;</td>
      <td>-0.230305</td>
    </tr>
    <tr>
      <th>2</th>
      <td>&gt;</td>
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>
```

	a	b
0	&	-0.474063
1	<	-0.230305
2	>	-0.400654

Not escaped:

```
In [313]: print(df.to_html(escape=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
```

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

	a	b
0	&	-0.474063
1	<	-0.230305
2	>	-0.400654

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

HTML Table Parsing Gotchas

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

Issues with lxml

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

Issues with BeautifulSoup4 using lxml as a backend

Scroll To Top

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

Issues with BeautifulSoup4 using html5lib as a backend

- Benefits
  - **html5lib** is far more lenient than **lxml** and consequently deals with *real-life markup* in a much saner way rather than just, e.g., dropping an element without notifying you.
  - **html5lib** *generates valid HTML5 markup from invalid markup automatically*. This is extremely important for parsing HTML tables, since it guarantees a valid document. However, that does NOT mean that it is "correct", since the process of fixing markup does not have a single definition.
  - **html5lib** is pure Python and requires no additional build steps beyond its own installation.
- Drawbacks
  - The biggest drawback to using **html5lib** is that it is slow as molasses. However consider the fact that many tables on the web are not big enough for the parsing algorithm runtime to matter. It is more likely that the bottleneck will be in the process of reading the raw text from the URL over the web, i.e., IO (input-output). For very large tables, this might not be true.

Excel files

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

Reading Excel files

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

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

`ExcelFile` class

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

Scroll To Top

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

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

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

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

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

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

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

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

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

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

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

Specifying sheets

Scroll To Top

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

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

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

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

Using the sheet index:

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

Using all default values:

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

Using None to get all sheets:

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

Using a list to get multiple sheets:

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

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

Reading a `MultiIndex`

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

Scroll To Top

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

```
In [314]: df = pd.DataFrame({'a': [1, 2, 3, 4], 'b': [5, 6, 7, 8]},
   .....:                   index=pd.MultiIndex.from_product([['a', 'b'], ['c', 'd']]))
   .....:

In [315]: df.to_excel('path_to_file.xlsx')

In [316]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1])

In [317]: df
Out[317]:
     a  b
a c  1  5
  d  2  6
b c  3  7
  d  4  8
```

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

```
In [318]: df.index = df.index.set_names(['lvl1', 'lvl2'])

In [319]: df.to_excel('path_to_file.xlsx')

In [320]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1])

In [321]: df
Out[321]:
           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 [322]: df.columns = pd.MultiIndex.from_product([['a'], ['b', 'd']],
   .....:                                         names=['c1', 'c2'])
   .....:

In [323]: df.to_excel('path_to_file.xlsx')

In [324]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1], header=[0, 1])

In [325]: df
Out[325]:
c1         a
c2         b  d
lvl1 lvl2
a    c     1  5
     d     2  6
b    c     3  7
     d     4  8
```

Scroll To Top

Parsing specific columns

It is often the case that users will insert columns to do temporary computations in Excel and you may not want to read in those columns. `read_excel` takes a `usecols` keyword to allow you to specify a subset of columns to parse.

Deprecated since version 0.24.0.

Passing in an integer for `usecols` has been deprecated. Please pass in a list of ints from 0 to `usecols` inclusive instead.

If `usecols` is an integer, then it is assumed to indicate the last column to be parsed.

```
pd.read_excel('path_to_file.xls', 'Sheet1', usecols=2)
```

You can also 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]`.

New in version 0.24.

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']`.

New in version 0.24.

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

Scroll To Top

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

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

New in version 0.20.

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:

Scroll To Top

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

Files with a `.xls` extension will be written using `xlwt` and those with a `.xlsx` extension will be written using `xlsxwriter` (if available) or `openpyxl`.

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')
```

Note: Wringing a little more performance out of `read_excel` Internally, Excel stores all numeric data as floats. Because this can produce unexpected behavior when reading in data, pandas defaults to trying to convert integers to floats if it doesn't lose information (`1.0 --> 1`). You can pass `convert_float=False` to disable this behavior, which may give a slight performance improvement.

Writing Excel files to memory

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

```
# Safe import for either Python 2.x or 3.x
try:
    from io import BytesIO
except ImportError:
    from cStringIO import StringIO as 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()
```

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.

Scroll To Top

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`, and xlwt for `.xls` files. 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`
- `xlwt`

```
# 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 parameters on the `DataFrame`'s `to_excel` method.

- `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

Scroll To Top

New in version 0.25.

The **read_excel()** method can also read OpenDocument spreadsheets using the `odfpy` module. The semantics and features for reading OpenDocument spreadsheets match what can be done for Excel files using `engine='odf'`.

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

Note: Currently pandas only supports *reading* OpenDocument spreadsheets. Writing is not implemented.

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.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [4, 5, 6],
...                    'C': ['p', 'q', 'r']},
...                   index=['x', 'y', 'z'])
>>> 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
```

Scroll To Top

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 [326]: df
Out[326]: 
c1         a   
c2         b  d
lvl1 lvl2      
a    c     1  5
     d     2  6
b    c     3  7
     d     4  8

In [327]: 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 [328]: pd.read_pickle('foo.pkl')
Out[328]: 
c1         a   
c2         b  d
lvl1 lvl2      
a    c     1  5
     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 pandas version 0.20.3

Compressed pickle files

New in version 0.20.0.

Scroll To Top

`read_pickle()`, `DataFrame.to_pickle()` and `Series.to_pickle()` can read and write compressed pickle files. The compression types of `gzip`, `bz2`, `xz` 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`, or `xz` if filename ends in `'.gz'`, `'.bz2'`, `'.zip'`, or `'.xz'`, respectively.

```
In [329]: df = pd.DataFrame({
   .....:     'A': np.random.randn(1000),
   .....:     'B': 'foo',
   .....:     'C': pd.date_range('20130101', periods=1000, freq='s')})
   .....:

In [330]: df
Out[330]:
            A    B                   C
0   -0.288267  foo 2013-01-01 00:00:00
1   -0.084905  foo 2013-01-01 00:00:01
2    0.004772  foo 2013-01-01 00:00:02
3    1.382989  foo 2013-01-01 00:00:03
4    0.343635  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.220893  foo 2013-01-01 00:16:35
996  0.492996  foo 2013-01-01 00:16:36
997 -0.461625  foo 2013-01-01 00:16:37
998  1.361779  foo 2013-01-01 00:16:38
999 -1.197988  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Using an explicit compression type:

```
In [331]: df.to_pickle("data.pkl.compress", compression="gzip")

In [332]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [333]: rt
Out[333]:
            A    B                   C
0   -0.288267  foo 2013-01-01 00:00:00
1   -0.084905  foo 2013-01-01 00:00:01
2    0.004772  foo 2013-01-01 00:00:02
3    1.382989  foo 2013-01-01 00:00:03
4    0.343635  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.220893  foo 2013-01-01 00:16:35
996  0.492996  foo 2013-01-01 00:16:36
997 -0.461625  foo 2013-01-01 00:16:37
998  1.361779  foo 2013-01-01 00:16:38
999 -1.197988  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Inferring compression type from the extension:

```
In [334]: df.to_pickle("data.pkl.xz", compression="infer")

In [335]: rt = pd.read_pickle("data.pkl.xz", compression="infer")

In [336]: rt
```

Scroll To Top

```
Out[336]:
            A    B                   C
0   -0.288267  foo 2013-01-01 00:00:00
1   -0.084905  foo 2013-01-01 00:00:01
2    0.004772  foo 2013-01-01 00:00:02
3    1.382989  foo 2013-01-01 00:00:03
4    0.343635  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.220893  foo 2013-01-01 00:16:35
996  0.492996  foo 2013-01-01 00:16:36
997 -0.461625  foo 2013-01-01 00:16:37
998  1.361779  foo 2013-01-01 00:16:38
999 -1.197988  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

The default is to 'infer':

```
In [337]: df.to_pickle("data.pkl.gz")

In [338]: rt = pd.read_pickle("data.pkl.gz")

In [339]: rt
Out[339]:
            A    B                   C
0   -0.288267  foo 2013-01-01 00:00:00
1   -0.084905  foo 2013-01-01 00:00:01
2    0.004772  foo 2013-01-01 00:00:02
3    1.382989  foo 2013-01-01 00:00:03
4    0.343635  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.220893  foo 2013-01-01 00:16:35
996  0.492996  foo 2013-01-01 00:16:36
997 -0.461625  foo 2013-01-01 00:16:37
998  1.361779  foo 2013-01-01 00:16:38
999 -1.197988  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]

In [340]: df["A"].to_pickle("s1.pkl.bz2")

In [341]: rt = pd.read_pickle("s1.pkl.bz2")

In [342]: rt
Out[342]:
0     -0.288267
1     -0.084905
2      0.004772
3      1.382989
4      0.343635
         ...
995   -0.220893
996    0.492996
997   -0.461625
998    1.361779
999   -1.197988
Name: A, Length: 1000, dtype: float64
```

Scroll To Top

msgpack

pandas supports the `msgpack` format for object serialization. This is a lightweight portable binary format, similar to binary JSON, that is highly space efficient, and provides good performance both on the writing (serialization), and reading (deserialization).

Warning: The msgpack format is deprecated as of 0.25 and will be removed in a future version. It is recommended to use pyarrow for on-the-wire transmission of pandas objects.

Warning: `read_msgpack()` is only guaranteed backwards compatible back to pandas version 0.20.3

```
In [343]: df = pd.DataFrame(np.random.rand(5, 2), columns=list('AB'))

In [344]: df.to_msgpack('foo.msg')

In [345]: pd.read_msgpack('foo.msg')
Out[345]: 
          A         B
0  0.275432  0.293583
1  0.842639  0.165381
2  0.608925  0.778891
3  0.136543  0.029703
4  0.318083  0.604870

In [346]: s = pd.Series(np.random.rand(5), index=pd.date_range('20130101', periods=5))
```

You can pass a list of objects and you will receive them back on deserialization.

```
In [347]: pd.to_msgpack('foo.msg', df, 'foo', np.array([1, 2, 3]), s)

In [348]: pd.read_msgpack('foo.msg')
Out[348]: 
[          A         B
 0  0.275432  0.293583
 1  0.842639  0.165381
 2  0.608925  0.778891
 3  0.136543  0.029703
 4  0.318083  0.604870, 'foo', array([1, 2, 3]), 2013-01-01    0.330824
 2013-01-02    0.790825
 2013-01-03    0.308468
 2013-01-04    0.092397
 2013-01-05    0.703091
 Freq: D, dtype: float64]
```

You can pass `iterator=True` to iterate over the unpacked results:

```
In [349]: for o in pd.read_msgpack('foo.msg', iterator=True):
   .....:     print(o)
   .....: 
          A         B
0  0.275432  0.293583
1  0.842639  0.165381
2  0.608925  0.778891
3  0.136543  0.029703
4  0.318083  0.604870
foo
[1 2 3]
```

Scroll To Top

```
2013-01-01    0.330824
2013-01-02    0.790825
2013-01-03    0.308468
2013-01-04    0.092397
2013-01-05    0.703091
Freq: D, dtype: float64
```

You can pass `append=True` to the writer to append to an existing pack:

```
In [350]: df.to_msgpack('foo.msg', append=True)

In [351]: pd.read_msgpack('foo.msg')
Out[351]:
[          A         B
 0  0.275432  0.293583
 1  0.842639  0.165381
 2  0.608925  0.778891
 3  0.136543  0.029703
 4  0.318083  0.604870, 'foo', array([1, 2, 3]), 2013-01-01    0.330824
 2013-01-02    0.790825
 2013-01-03    0.308468
 2013-01-04    0.092397
 2013-01-05    0.703091
 Freq: D, dtype: float64,           A         B
 0  0.275432  0.293583
 1  0.842639  0.165381
 2  0.608925  0.778891
 3  0.136543  0.029703
 4  0.318083  0.604870]
```

Unlike other io methods, `to_msgpack` is available on both a per-object basis, `df.to_msgpack()` and using the top-level `pd.to_msgpack(...)` where you can pack arbitrary collections of Python lists, dicts, scalars, while intermixing pandas objects.

```
In [352]: pd.to_msgpack('foo2.msg', {'dict': [{'df': df}, {'string': 'foo'},
   .....:                                     {'scalar': 1.}, {'s': s}]})
   .....:

In [353]: pd.read_msgpack('foo2.msg')
Out[353]:
{'dict': ({'df':           A         B
   0  0.275432  0.293583
   1  0.842639  0.165381
   2  0.608925  0.778891
   3  0.136543  0.029703
   4  0.318083  0.604870},
  {'string': 'foo'},
  {'scalar': 1.0},
  {'s': 2013-01-01    0.330824
   2013-01-02    0.790825
   2013-01-03    0.308468
   2013-01-04    0.092397
   2013-01-05    0.703091
   Freq: D, dtype: float64})}
```

Scroll To Top

Read/write API

Msgpacks can also be read from and written to strings.

```
In [354]: df.to_msgpack()
Out[354]: b'\x84\xa3typ\xadblock_manager\xa5klass\xa9DataFrame\xa4axes\x92\x86\xa3typ\x
```

Furthermore you can concatenate the strings to produce a list of the original objects.

```
In [355]: pd.read_msgpack(df.to_msgpack() + s.to_msgpack())
Out[355]: 
[          A         B
 0  0.275432  0.293583
 1  0.842639  0.165381
 2  0.608925  0.778891
 3  0.136543  0.029703
 4  0.318083  0.604870, 2013-01-01    0.330824
 2013-01-02    0.790825
 2013-01-03    0.308468
 2013-01-04    0.092397
 2013-01-05    0.703091
 Freq: D, dtype: float64]
```

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 requires `PyTables` >= 3.0.0. There is a indexing bug in `PyTables` < 3.2 which may appear when querying stores using an index. If you see a subset of results being returned, upgrade to `PyTables` >= 3.2. Stores created previously will need to be rewritten using the updated version.

```
In [356]: store = pd.HDFStore('store.h5')

In [357]: 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 [358]: index = pd.date_range('1/1/2000', periods=8)

In [359]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [360]: df = pd.DataFrame(np.random.randn(8, 3), index=index,
   .....:                   columns=['A', 'B', 'C'])
   .....: 

# store.put('s', s) is an equivalent method
In [361]: store['s'] = s

In [362]: store['df'] = df
```

Scroll To Top

```
In [363]: store
Out[363]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

In a current or later Python session, you can retrieve stored objects:

```
# store.get('df') is an equivalent method
In [364]: store['df']
Out[364]:
                   A         B         C
2000-01-01 -0.426936 -1.780784  0.322691
2000-01-02  1.638174 -2.184251  0.049673
2000-01-03 -1.022803  0.889445  2.827717
2000-01-04  1.767446 -1.305266 -0.378355
2000-01-05  0.486743  0.954551  0.859671
2000-01-06 -1.170458 -1.211386 -0.852728
2000-01-07 -0.450781  1.064650  1.014927
2000-01-08 -0.810399  0.254343 -0.875526

# dotted (attribute) access provides get as well
In [365]: store.df
Out[365]:
                   A         B         C
2000-01-01 -0.426936 -1.780784  0.322691
2000-01-02  1.638174 -2.184251  0.049673
2000-01-03 -1.022803  0.889445  2.827717
2000-01-04  1.767446 -1.305266 -0.378355
2000-01-05  0.486743  0.954551  0.859671
2000-01-06 -1.170458 -1.211386 -0.852728
2000-01-07 -0.450781  1.064650  1.014927
2000-01-08 -0.810399  0.254343 -0.875526
```

Deletion of the object specified by the key:

```
# store.remove('df') is an equivalent method
In [366]: del store['df']

In [367]: store
Out[367]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Closing a Store and using a context manager:

```
In [368]: store.close()

In [369]: store
Out[369]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

In [370]: store.is_open
Out[370]: False

# Working with, and automatically closing the store using a context manager
In [371]: with pd.HDFStore('store.h5') as store:
```

Scroll To Top

```
   .....:     store.keys()
   .....:
```

Read/write API

`HDFStore` supports an top-level API using `read_hdf` for reading and `to_hdf` for writing, similar to how `read_csv` and `to_csv` work.

```
In [372]: df_tl = pd.DataFrame({'A': list(range(5)), 'B': list(range(5))})

In [373]: df_tl.to_hdf('store_tl.h5', 'table', append=True)

In [374]: pd.read_hdf('store_tl.h5', 'table', where=['index>2'])
Out[374]: 
   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 [375]: df_with_missing = pd.DataFrame({'col1': [0, np.nan, 2],
   .....:                                 'col2': [1, np.nan, np.nan]})
   .....: 

In [376]: df_with_missing
Out[376]: 
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [377]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
   .....:                        format='table', mode='w')
   .....: 

In [378]: pd.read_hdf('file.h5', 'df_with_missing')
Out[378]: 
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [379]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
   .....:                        format='table', mode='w', dropna=True)
   .....: 

In [380]: pd.read_hdf('file.h5', 'df_with_missing')
Out[380]: 
   col1  col2
0   0.0   1.0
2   2.0   NaN
```

Scroll To Top

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'`.

> **Warning:** A `fixed` format will raise a `TypeError` if you try to retrieve using a `where`:
>
> ```
> >>> pd.DataFrame(np.random.randn(10, 2)).to_hdf('test_fixed.h5', 'df')
> >>> pd.read_hdf('test_fixed.h5', 'df', where='index>5')
> TypeError: cannot pass a where specification when reading a fixed format.
>            this store must be selected in its entirety
> ```

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 [381]: store = pd.HDFStore('store.h5')

In [382]: df1 = df[0:4]

In [383]: df2 = df[4:]

# append data (creates a table automatically)
In [384]: store.append('df', df1)

In [385]: store.append('df', df2)

In [386]: store
Out[386]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# select the entire object
In [387]: store.select('df')
Out[387]:
                   A         B         C
2000-01-01 -0.426936 -1.780784  0.322691
2000-01-02  1.638174 -2.184251  0.049673
2000-01-03 -1.022803  0.889445  2.827717
2000-01-04  1.767446 -1.305266 -0.378355
2000-01-05  0.486743  0.954551  0.859671
2000-01-06 -1.170458 -1.211386 -0.852728
2000-01-07 -0.450781  1.064650  1.014927
2000-01-08 -0.810399  0.254343 -0.875526
```

Scroll To Top

```
# the type of stored data
In [388]: store.root.df._v_attrs.pandas_type
Out[388]: '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 with out 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 [389]: store.put('foo/bar/bah', df)

In [390]: store.append('food/orange', df)

In [391]: store.append('food/apple', df)

In [392]: store
Out[392]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# a list of keys are returned
In [393]: store.keys()
Out[393]: ['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [394]: store.remove('food')

In [395]: store
Out[395]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

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.

New in version 0.24.0.

```
In [396]: 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 -0.426936 -1.780784  0.322691
```

Scroll To Top

```
2000-01-02  1.638174 -2.184251  0.049673
2000-01-03 -1.022803  0.889445  2.827717
2000-01-04  1.767446 -1.305266 -0.378355
2000-01-05  0.486743  0.954551  0.859671
2000-01-06 -1.170458 -1.211386 -0.852728
2000-01-07 -0.450781  1.064650  1.014927
2000-01-08 -0.810399  0.254343 -0.875526
GROUP: /foo/bar
KEY: /foo/bar/bah
                   A         B         C
2000-01-01 -0.426936 -1.780784  0.322691
2000-01-02  1.638174 -2.184251  0.049673
2000-01-03 -1.022803  0.889445  2.827717
2000-01-04  1.767446 -1.305266 -0.378355
2000-01-05  0.486743  0.954551  0.859671
2000-01-06 -1.170458 -1.211386 -0.852728
2000-01-07 -0.450781  1.064650  1.014927
2000-01-08 -0.810399  0.254343 -0.875526
```

Warning: Hierarchical keys cannot be retrieved as dotted (attribute) access as described above for items stored under the root node.

```
In [8]: store.foo.bar.bah
AttributeError: 'HDFStore' object has no attribute 'foo'

# you can directly access the actual PyTables node but using the root node
In [9]: store.root.foo.bar.bah
Out[9]:
/foo/bar/bah (Group) ''
  children := ['block0_items' (Array), 'block0_values' (Array), 'axis0' (Array), 'axi
```

Instead, use explicit string based keys:

```
In [397]: store['foo/bar/bah']
Out[397]:
                   A         B         C
2000-01-01 -0.426936 -1.780784  0.322691
2000-01-02  1.638174 -2.184251  0.049673
2000-01-03 -1.022803  0.889445  2.827717
2000-01-04  1.767446 -1.305266 -0.378355
2000-01-05  0.486743  0.954551  0.859671
2000-01-06 -1.170458 -1.211386 -0.852728
2000-01-07 -0.450781  1.064650  1.014927
2000-01-08 -0.810399  0.254343 -0.875526
```

Storing types

Storing mixed types in a table

Scroll To Top

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 [398]: 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 [399]: df_mixed.loc[df_mixed.index[3:5],
   .....:              ['A', 'B', 'string', 'datetime64']] = np.nan
   .....: 

In [400]: store.append('df_mixed', df_mixed, min_itemsize={'values': 50})

In [401]: df_mixed1 = store.select('df_mixed')

In [402]: df_mixed1
Out[402]: 
          A         B         C  string  int  bool datetime64
0 -0.980856  0.298656  0.151508  string    1  True 2001-01-02
1 -0.906920 -1.294022  0.587939  string    1  True 2001-01-02
2  0.988185 -0.618845  0.043096  string    1  True 2001-01-02
3       NaN       NaN  0.362451     NaN    1  True        NaT
4       NaN       NaN  1.356269     NaN    1  True        NaT
5 -0.772889 -0.340872  1.798994  string    1  True 2001-01-02
6 -0.043509 -0.303900  0.567265  string    1  True 2001-01-02
7  0.768606 -0.871948 -0.044348  string    1  True 2001-01-02

In [403]: df_mixed1.dtypes.value_counts()
Out[403]: 
float64           2
float32           1
datetime64[ns]    1
int64             1
bool              1
object            1
dtype: int64

# we have provided a minimum string column size
In [404]: store.root.df_mixed.table
Out[404]: 
/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": Int64Col(shape=(1,), dflt=0, 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": StringCol(itemsize=50, shape=(1,), dflt=b'', pos=6)}
  byteorder := 'little'
  chunkshape := (689,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

Scroll To Top

Storing MultiIndex DataFrames

Storing MultiIndex `DataFrames` as tables is very similar to storing/selecting from homogeneous index `DataFrames`.

```
In [405]: 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 [406]: df_mi = pd.DataFrame(np.random.randn(10, 3), index=index,
   .....:                      columns=['A', 'B', 'C'])
   .....:

In [407]: df_mi
Out[407]: 
                  A         B         C
foo bar                              
foo one    0.031885  0.641045  0.479460
    two   -0.630652 -0.182400 -0.789979
    three -0.282700 -0.813404  1.252998
bar one    0.758552  0.384775 -1.133177
    two   -1.002973 -1.644393 -0.311536
baz two   -0.615506 -0.084551 -1.318575
    three  0.923929 -0.105981  0.429424
qux one   -1.034590  0.542245 -0.384429
    two    0.170697 -0.200289  1.220322
    three -1.001273  0.162172  0.376816

In [408]: store.append('df_mi', df_mi)

In [409]: store.select('df_mi')
Out[409]: 
                  A         B         C
foo bar                              
foo one    0.031885  0.641045  0.479460
    two   -0.630652 -0.182400 -0.789979
    three -0.282700 -0.813404  1.252998
bar one    0.758552  0.384775 -1.133177
    two   -1.002973 -1.644393 -0.311536
baz two   -0.615506 -0.084551 -1.318575
    three  0.923929 -0.105981  0.429424
qux one   -1.034590  0.542245 -0.384429
    two    0.170697 -0.200289  1.220322
    three -1.001273  0.162172  0.376816

# the levels are automatically included as data columns
In [410]: store.select('df_mi', 'foo=bar')
Out[410]: 
                A         B         C
foo bar                            
bar one  0.758552  0.384775 -1.133177
    two -1.002973 -1.644393 -0.311536
```

Scroll To Top

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.

A query is specified using the `Term` class under the hood, as a boolean expression.

- `index` and `columns` are supported indexers of a `DataFrames`.
- if `data_columns` are specified, these can be used as additional indexers.

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`

The right-hand side of the sub-expression (after a comparison operator) can be:

Scroll To Top

- 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', 'index == %s' % 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 [411]: dfq = pd.DataFrame(np.random.randn(10, 4), columns=list('ABCD'),
   .....:                    index=pd.date_range('20130101', periods=10))
   .....:

In [412]: store.append('dfq', dfq, format='table', data_columns=True)
```

Use boolean expressions, with in-line function evaluation.

```
In [413]: store.select('dfq', "index>pd.Timestamp('20130104') & columns=['A', 'B']")
Out[413]:
                   A         B
2013-01-05  0.450263  0.755221
2013-01-06  0.019915  0.300003
2013-01-07  1.878479 -0.026513
2013-01-08  3.272320  0.077044
2013-01-09 -0.398346  0.507286
2013-01-10  0.516017 -0.501550
```

Scroll To Top

Use and inline column reference

```
In [414]: store.select('dfq', where="A>0 or C>0")
Out[414]:
                   A         B         C         D
2013-01-01 -0.161614 -1.636805  0.835417  0.864817
2013-01-02  0.843452 -0.122918 -0.026122 -1.507533
2013-01-03  0.335303 -1.340566 -1.024989  1.125351
2013-01-05  0.450263  0.755221 -1.506656  0.808794
2013-01-06  0.019915  0.300003 -0.727093 -1.119363
2013-01-07  1.878479 -0.026513  0.573793  0.154237
2013-01-08  3.272320  0.077044  0.397034 -0.613983
2013-01-10  0.516017 -0.501550  0.138212  0.218366
```

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 [415]: store.select('df', "columns=['A', 'B']")
Out[415]:
                   A         B
2000-01-01 -0.426936 -1.780784
2000-01-02  1.638174 -2.184251
2000-01-03 -1.022803  0.889445
2000-01-04  1.767446 -1.305266
2000-01-05  0.486743  0.954551
2000-01-06 -1.170458 -1.211386
2000-01-07 -0.450781  1.064650
2000-01-08 -0.810399  0.254343
```

`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.

> **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.

Using 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 [416]: from datetime import timedelta

In [417]: dftd = pd.DataFrame({'A': pd.Timestamp('20130101'),
   .....:                      'B': [pd.Timestamp('20130101') + timedelta(days=i,
   .....:                                                                  seconds=10)
   .....:                            for i in range(10)]})
   .....:
```

Scroll To Top

```
In [418]: dftd['C'] = dftd['A'] - dftd['B']

In [419]: dftd
Out[419]: 
           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 [420]: store.append('dftd', dftd, data_columns=True)

In [421]: store.select('dftd', "C<'-3.5D'")
Out[421]: 
           A                   B                  C
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
```

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 [422]: i = store.root.df.table.cols.index.index

In [423]: i.optlevel, i.kind
Out[423]: (6, 'medium')

# change an index by passing new parameters
In [424]: store.create_table_index('df', optlevel=9, kind='full')

In [425]: i = store.root.df.table.cols.index.index

In [426]: i.optlevel, i.kind
Out[426]: (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.

Scroll To Top

```
In [427]: df_1 = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [428]: df_2 = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [429]: st = pd.HDFStore('appends.h5', mode='w')

In [430]: st.append('df', df_1, data_columns=['B'], index=False)

In [431]: st.append('df', df_2, data_columns=['B'], index=False)

In [432]: st.get_storer('df').table
Out[432]: 
/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,)
```

Then create the index when finished appending.

```
In [433]: st.create_table_index('df', columns=['B'], optlevel=9, kind='full')

In [434]: st.get_storer('df').table
Out[434]: 
/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, full, shuffle, zlib(1)).is_csi=True}

In [435]: 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 [436]: df_dc = df.copy()

In [437]: df_dc['string'] = 'foo'

In [438]: df_dc.loc[df_dc.index[4:6], 'string'] = np.nan

In [439]: df_dc.loc[df_dc.index[7:9], 'string'] = 'bar'
```

Scroll To Top

```
In [440]: df_dc['string2'] = 'cool'

In [441]: df_dc.loc[df_dc.index[1:3], ['B', 'C']] = 1.0

In [442]: df_dc
Out[442]: 
                   A         B         C string string2
2000-01-01 -0.426936 -1.780784  0.322691    foo    cool
2000-01-02  1.638174  1.000000  1.000000    foo    cool
2000-01-03 -1.022803  1.000000  1.000000    foo    cool
2000-01-04  1.767446 -1.305266 -0.378355    foo    cool
2000-01-05  0.486743  0.954551  0.859671    NaN    cool
2000-01-06 -1.170458 -1.211386 -0.852728    NaN    cool
2000-01-07 -0.450781  1.064650  1.014927    foo    cool
2000-01-08 -0.810399  0.254343 -0.875526    bar    cool

# on-disk operations
In [443]: store.append('df_dc', df_dc, data_columns=['B', 'C', 'string', 'string2'])

In [444]: store.select('df_dc', where='B > 0')
Out[444]: 
                   A         B         C string string2
2000-01-02  1.638174  1.000000  1.000000    foo    cool
2000-01-03 -1.022803  1.000000  1.000000    foo    cool
2000-01-05  0.486743  0.954551  0.859671    NaN    cool
2000-01-07 -0.450781  1.064650  1.014927    foo    cool
2000-01-08 -0.810399  0.254343 -0.875526    bar    cool

# getting creative
In [445]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')
Out[445]: 
                   A        B         C string string2
2000-01-02  1.638174  1.00000  1.000000    foo    cool
2000-01-03 -1.022803  1.00000  1.000000    foo    cool
2000-01-07 -0.450781  1.06465  1.014927    foo    cool

# this is in-memory version of this type of selection
In [446]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]
Out[446]: 
                   A        B         C string string2
2000-01-02  1.638174  1.00000  1.000000    foo    cool
2000-01-03 -1.022803  1.00000  1.000000    foo    cool
2000-01-07 -0.450781  1.06465  1.014927    foo    cool

# we have automagically created this index and the B/C/string/string2
# columns are stored separately as ``PyTables`` columns
In [447]: store.root.df_dc.table
Out[447]: 
/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, medium, shuffle, zlib(1)).is_csi=False,
    "B": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "C": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string2": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

Scroll To Top

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!).

Iterator

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 [448]: for df in store.select('df', chunksize=3):
   .....:     print(df)
   .....:
                   A         B         C
2000-01-01 -0.426936 -1.780784  0.322691
2000-01-02  1.638174 -2.184251  0.049673
2000-01-03 -1.022803  0.889445  2.827717
                   A         B         C
2000-01-04  1.767446 -1.305266 -0.378355
2000-01-05  0.486743  0.954551  0.859671
2000-01-06 -1.170458 -1.211386 -0.852728
                   A         B         C
2000-01-07 -0.450781  1.064650  1.014927
2000-01-08 -0.810399  0.254343 -0.875526
```

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 [449]: dfeq = pd.DataFrame({'number': np.arange(1, 11)})

In [450]: dfeq
Out[450]:
   number
0       1
1       2
2       3
3       4
4       5
5       6
6       7
7       8
8       9
```

Scroll To Top

```
9      10

In [451]: store.append('dfeq', dfeq, data_columns=['number'])

In [452]: def chunks(l, n):
   .....:     return [l[i:i + n] for i in range(0, len(l), n)]
   .....:

In [453]: evens = [2, 4, 6, 8, 10]

In [454]: coordinates = store.select_as_coordinates('dfeq', 'number=evens')

In [455]: 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 [456]: store.select_column('df_dc', 'index')
Out[456]:
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 [457]: store.select_column('df_dc', 'string')
Out[457]:
0    foo
1    foo
2    foo
3    foo
4    NaN
5    NaN
6    foo
7    bar
Name: string, dtype: object
```

Scroll To Top

Selecting coordinates

Sometimes you want to get the coordinates (a.k.a the index locations) of your query. This returns an `Int64Index` of the resulting locations. These coordinates can also be passed to subsequent `where` operations.

```
In [458]: df_coord = pd.DataFrame(np.random.randn(1000, 2),
   .....:                         index=pd.date_range('20000101', periods=1000))
   .....:

In [459]: store.append('df_coord', df_coord)

In [460]: c = store.select_as_coordinates('df_coord', 'index > 20020101')

In [461]: c
Out[461]:
Int64Index([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 [462]: store.select('df_coord', where=c)
Out[462]:
                   0         1
2002-01-02  0.440865 -0.151651
2002-01-03 -1.195089  0.285093
2002-01-04 -0.925046  0.386081
2002-01-05 -1.942756  0.277699
2002-01-06  0.811776  0.528965
...              ...       ...
2002-09-22  1.061729  0.618085
2002-09-23 -0.209744  0.677197
2002-09-24 -1.808184  0.185667
2002-09-25 -0.208629  0.928603
2002-09-26  1.579717 -1.259530

[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.

```
In [463]: df_mask = pd.DataFrame(np.random.randn(1000, 2),
   .....:                        index=pd.date_range('20000101', periods=1000))
   .....:

In [464]: store.append('df_mask', df_mask)

In [465]: c = store.select_column('df_mask', 'index')

In [466]: where = c[pd.DatetimeIndex(c).month == 5].index

In [467]: store.select('df_mask', where=where)
Out[467]:
                   0         1
2000-05-01 -1.199892  1.073701
2000-05-02 -1.058552  0.658487
```

Scroll To Top

```
2000-05-03 -0.015418  0.452879
2000-05-04  1.737818  0.426356
2000-05-05 -0.711668 -0.021266
...              ...       ...
2002-05-27  0.656196  0.993383
2002-05-28 -0.035399 -0.269286
2002-05-29  0.704503  2.574402
2002-05-30 -1.301443  2.770770
2002-05-31 -0.807599  0.420431

[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 [468]: store.get_storer('df_dc').nrows
Out[468]: 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.

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 [469]: 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 [470]: df_mt['foo'] = 'bar'

In [471]: df_mt.loc[df_mt.index[1], ('A', 'B')] = np.nan

# you can also create the tables individually
In [472]: store.append_to_multiple({'df1_mt': ['A', 'B'], 'df2_mt': None},
```

Scroll To Top

```
   .....:                              df_mt, selector='df1_mt')
   .....:

In [473]: store
Out[473]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# individual tables were created
In [474]: store.select('df1_mt')
Out[474]:
                   A         B
2000-01-01  0.475158  0.427905
2000-01-02       NaN       NaN
2000-01-03 -0.201829  0.651656
2000-01-04 -0.766427 -1.852010
2000-01-05  1.642910 -0.055583
2000-01-06  0.187880  1.536245
2000-01-07 -1.801014  0.244721
2000-01-08  3.055033 -0.683085

In [475]: store.select('df2_mt')
Out[475]:
                   C         D         E         F  foo
2000-01-01  1.846285 -0.044826  0.074867  0.156213  bar
2000-01-02  0.446978 -0.323516  0.311549 -0.661368  bar
2000-01-03 -2.657254  0.649636  1.520717  1.604905  bar
2000-01-04 -0.201100 -2.107934 -0.450691 -0.748581  bar
2000-01-05  0.543779  0.111444  0.616259 -0.679614  bar
2000-01-06  0.831475 -0.566063  1.130163 -1.004539  bar
2000-01-07  0.745984  1.532560  0.229376  0.526671  bar
2000-01-08 -0.922301  2.760888  0.515474 -0.129319  bar

# as a multiple
In [476]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'],
   .....:                          selector='df1_mt')
   .....:
Out[476]:
                   A         B         C         D         E         F  foo
2000-01-01  0.475158  0.427905  1.846285 -0.044826  0.074867  0.156213  bar
2000-01-06  0.187880  1.536245  0.831475 -0.566063  1.130163 -1.004539  bar
```

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

Scroll To Top

- 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 removed, then the following data moved. On the other hand a delete operation on the `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.

New in version 0.20.2: 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.
- blosc:zlib: A classic; somewhat slower than the previous ones, but achieving better compression ratios.

Scroll To Top

- 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

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 (GH2397) 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

Scroll To Top

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	
`datetime64[ns]`	`NaT`
`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 [477]: dfcat = pd.DataFrame({'A': pd.Series(list('aabbcdba')).astype('category'),
   .....:                       'B': np.random.randn(8)})
   .....:

In [478]: dfcat
Out[478]:
   A         B
0  a  1.706605
1  a  1.373485
2  b -0.758424
3  b -0.116984
4  c -0.959461
5  d -1.517439
6  b -0.453150
7  a -0.827739

In [479]: dfcat.dtypes
Out[479]:
A    category
B     float64
dtype: object
```

Scroll To Top

```
In [480]: cstore = pd.HDFStore('cats.h5', mode='w')

In [481]: cstore.append('dfcat', dfcat, format='table', data_columns=['A'])

In [482]: result = cstore.select('dfcat', where="A in ['b', 'c']")

In [483]: result
Out[483]:
   A         B
2  b -0.758424
3  b -0.116984
4  c -0.959461
6  b -0.453150

In [484]: result.dtypes
Out[484]:
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 [485]: dfs = pd.DataFrame({'A': 'foo', 'B': 'bar'}, index=list(range(5)))

In [486]: dfs
Out[486]:
     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 [487]: store.append('dfs', dfs, min_itemsize=30)

In [488]: store.get_storer('dfs').table
```

Scroll To Top

```
Out[488]:
/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, medium, shuffle, zlib(1)).is_csi=False}

# A is created as a data_column with a size of 30
# B is size is calculated
In [489]: store.append('dfs2', dfs, min_itemsize={'A': 30})

In [490]: store.get_storer('dfs2').table
Out[490]:
/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, medium, shuffle, zlib(1)).is_csi=False,
    "A": Index(6, medium, shuffle, 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 [491]: dfss = pd.DataFrame({'A': ['foo', 'bar', 'nan']})

In [492]: dfss
Out[492]:
     A
0  foo
1  bar
2  nan

In [493]: store.append('dfss', dfss)

In [494]: store.select('dfss')
Out[494]:
     A
0  foo
1  bar
2  NaN

# here you need to specify a different nan rep
In [495]: store.append('dfss2', dfss, nan_rep='_nan_')

In [496]: store.select('dfss2')
Out[496]:
     A
0  foo
1  bar
2  nan
```

Scroll To Top

External compatibility

`HDFStore` writes `table` format objects in specific formats suitable for producing loss-less round trips to pandas objects. For external compatibility, `HDFStore` can read native `PyTables` format tables.

It is possible to write an `HDFStore` object that can easily be imported into `R` using the `rhdf5` library (Package website). Create a table format store like this:

```
In [497]: df_for_r = pd.DataFrame({"first": np.random.rand(100),
   .....:                          "second": np.random.rand(100),
   .....:                          "class": np.random.randint(0, 2, (100, ))},
   .....:                         index=range(100))
   .....:

In [498]: df_for_r.head()
Out[498]:
      first    second  class
0  0.366979  0.794525      0
1  0.296639  0.635178      1
2  0.395751  0.359693      0
3  0.484648  0.970016      1
4  0.810047  0.332303      0

In [499]: store_export = pd.HDFStore('export.h5')

In [500]: store_export.append('df_for_r', df_for_r, data_columns=df_dc.columns)

In [501]: store_export
Out[501]:
<class 'pandas.io.pytables.HDFStore'>
File path: export.h5
```

In R this file can be read into a `data.frame` object using the `rhdf5` library. The following example function reads the corresponding column names and data values from the values and assembles them into a `data.frame`:

```
# Load values and column names for all datasets from corresponding nodes and
# insert them into one data.frame object.

library(rhdf5)

loadhdf5data <- function(h5File) {

listing <- h5ls(h5File)
# Find all data nodes, values are stored in *_values and corresponding column
# titles in *_items
data_nodes <- grep("_values", listing$name)
name_nodes <- grep("_items", listing$name)
data_paths = paste(listing$group[data_nodes], listing$name[data_nodes], sep = "/")
name_paths = paste(listing$group[name_nodes], listing$name[name_nodes], sep = "/")
columns = list()
for (idx in seq(data_paths)) {
  # NOTE: matrices returned by h5read have to be transposed to obtain
  # required Fortran order!
  data <- data.frame(t(h5read(h5File, data_paths[idx])))
  names <- t(h5read(h5File, name_paths[idx]))
  entry <- data.frame(data)
  colnames(entry) <- names
```

Scroll To Top

```
  columns <- append(columns, entry)
}

data <- data.frame(columns)

return(data)
}
```

Now you can import the `DataFrame` into R:

```
> data = loadhdf5data("transfer.hdf5")
> head(data)
         first    second class
1 0.4170220047 0.3266449     0
2 0.7203244934 0.5270581     0
3 0.0001143748 0.8859421     1
4 0.3023325726 0.3572698     1
5 0.1467558908 0.9085352     1
6 0.0923385948 0.6233601     1
```

Note: The R function lists the entire HDF5 file's contents and assembles the `data.frame` object from all matching nodes, so use this only as a starting point if you have stored multiple `DataFrame` objects to a single HDF5 file.

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 expected rows that `PyTables` will expected. 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

New in version 0.20.0.

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.

Scroll To Top

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.

- This is a newer library, and the format, though stable, is not guaranteed to be backward compatible to the earlier versions.
- 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
- Non supported types include `Period` and actual Python object types. These will raise a helpful error message on an attempt at serialization.

See the Full Documentation.

```
In [502]: 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 [503]: df
Out[503]: 
   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 [504]: df.dtypes
Out[504]: 
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 [505]: df.to_feather('example.feather')
```

Read from a feather file.

```
In [506]: result = pd.read_feather('example.feather')

In [507]: result
Out[507]: 
   a  b  c    d      e  f          g                         h
```

Scroll To Top

```
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 [508]: result.dtypes
Out[508]:
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

New in version 0.21.0.

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.
- Categorical dtypes can be serialized to parquet, but will de-serialize as `object` dtype.
- Non supported types include `Period` and actual Python object types. These will raise a helpful error message on an attempt at serialization.

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. Currently `pyarrow` does not support timedelta data, `fastparquet>=0.1.4` supports timezone aware datetimes.

Scroll To Top

These libraries differ by having different underlying dependencies (`fastparquet` by using `numba`, while `pyarrow` uses a c-library).

```
In [509]: 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')
   .....:

In [510]: df
Out[510]:
   a  b  c    d      e          f                         g
0  a  1  3  4.0   True 2013-01-01 2013-01-01 00:00:00-05:00
1  b  2  4  5.0  False 2013-01-02 2013-01-02 00:00:00-05:00
2  c  3  5  6.0   True 2013-01-03 2013-01-03 00:00:00-05:00

In [511]: df.dtypes
Out[511]:
a                        object
b                         int64
c                         uint8
d                       float64
e                          bool
f                datetime64[ns]
g    datetime64[ns, US/Eastern]
dtype: object
```

Write to a parquet file.

```
In [512]: df.to_parquet('example_pa.parquet', engine='pyarrow')

In [513]: df.to_parquet('example_fp.parquet', engine='fastparquet')
```

Read from a parquet file.

```
In [514]: result = pd.read_parquet('example_fp.parquet', engine='fastparquet')

In [515]: result = pd.read_parquet('example_pa.parquet', engine='pyarrow')

In [516]: result.dtypes
Out[516]:
a                        object
b                         int64
c                         uint8
d                       float64
e                          bool
f                datetime64[ns]
g    datetime64[ns, US/Eastern]
dtype: object
```

Scroll To Top

Read only certain columns of a parquet file.

```
In [517]: result = pd.read_parquet('example_fp.parquet',
   .....:                          engine='fastparquet', columns=['a', 'b'])
   .....:

In [518]: result = pd.read_parquet('example_pa.parquet',
   .....:                          engine='pyarrow', columns=['a', 'b'])
   .....:

In [519]: result.dtypes
Out[519]:
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 [520]: df = pd.DataFrame({'a': [1, 2], 'b': [3, 4]})

In [521]: 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()`**:

```
In [522]: 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

New in version 0.24.0.

Parquet supports partitioning of data based on the values of one or more columns.

```
In [523]: df = pd.DataFrame({'a': [0, 0, 1, 1], 'b': [0, 1, 0, 1]})

In [524]: df.to_parquet(fname='test', engine='pyarrow',
   .....:               partition_cols=['a'], compression=None)
   .....:
```

Scroll To Top

The *fname* 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
    └── ...
```

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. Database abstraction is provided by SQLAlchemy if installed. In addition you will need a driver library for your database. 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.

If SQLAlchemy is not installed, a fallback is only provided for sqlite (and for mysql for backwards compatibility, but this is deprecated and will be removed in a future version). This mode requires a Python database adapter which respect the Python DB-API.

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(self, 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 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

Scroll To Top

```
In [525]: from sqlalchemy import create_engine

# Create your engine.
In [526]: engine = create_engine('sqlite:///:memory:')
```

If you want to manage your own connections you can pass one of those instead:

```
with engine.connect() as conn, conn.begin():
    data = pd.read_sql_table('data', conn)
```

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 [527]: data
Out[527]:
   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 [528]: data.to_sql('data', engine)
```

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 [529]: data.to_sql('data_chunked', engine, chunksize=1000)
```

SQL data types

`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 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:

Scroll To Top

```
In [530]: from sqlalchemy.types import String

In [531]: data.to_sql('data_dtype', engine, dtype={'Col_1': String})
```

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.

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 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

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

New in version 0.24.0.

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 documention.
- 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:

Scroll To Top

```
# 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):
    # 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)
        else:
            table_name = table.name

        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 SQLAlchemy optional dependency installed.

```
In [532]: pd.read_sql_table('data', engine)
Out[532]:
   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
```

You can also specify the name of the column as the `DataFrame` index, and specify a subset of columns to be read.

```
In [533]: pd.read_sql_table('data', engine, index_col='id')
Out[533]:
    index       Date Col_1  Col_2  Col_3
id
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 [534]: pd.read_sql_table('data', engine, columns=['Col_1', 'Col_2'])
Out[534]:
  Col_1  Col_2
0     X  27.50
1     Y -12.50
2     Z   5.73
```

Scroll To Top

And you can explicitly force columns to be parsed as dates:

```
In [535]: pd.read_sql_table('data', engine, parse_dates=['Date'])
Out[535]:
   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('table', engine, schema='other_schema')
pd.read_sql_table('table', engine, schema='other_schema')
```

Querying

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 [536]: pd.read_sql_query('SELECT * FROM data', engine)
Out[536]:
   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 [537]: pd.read_sql_query("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;", engine)
Out[537]:
   id Col_1  Col_2
0  42     Y  -12.5
```

Scroll To Top

The **read_sql_query()** function supports a `chunksize` argument. Specifying this will return an iterator through chunks of the query result:

```
In [538]: df = pd.DataFrame(np.random.randn(20, 3), columns=list('abc'))

In [539]: df.to_sql('data_chunks', engine, index=False)
```

```
In [540]: for chunk in pd.read_sql_query("SELECT * FROM data_chunks",
   .....:                                engine, chunksize=5):
   .....:     print(chunk)
   .....:
          a         b         c
0 -0.900850 -0.323746  0.037100
1  0.057533 -0.032842  0.550902
2  1.026623  1.035455 -0.965140
3 -0.252405 -1.255987  0.639156
4  1.076701 -0.309155 -0.800182
          a         b         c
0 -0.206623  0.496077 -0.219935
1  0.631362 -1.166743  1.808368
2  0.023531  0.987573  0.471400
3 -0.982250 -0.192482  1.195452
4 -1.758855  0.477551  1.412567
          a         b         c
0 -1.120570  1.232764  0.417814
1  1.688089 -0.037645 -0.269582
2  0.646823 -0.603366  1.592966
3  0.724019 -0.515606 -0.180920
4  0.038244 -2.292866 -0.114634
          a         b         c
0 -0.970230 -0.963257 -0.128304
1  0.498621 -1.496506  0.701471
2 -0.272608 -0.119424 -0.882023
3 -0.253477  0.714395  0.664179
4  0.897140  0.455791  1.549590
```

You can also run a plain query without creating a `DataFrame` with **execute()**. This is useful for queries that don't return values, such as INSERT. This is functionally equivalent to calling `execute` on the SQLAlchemy engine or db connection object. Again, you must use the SQL syntax variant appropriate for your database.

```
from pandas.io import sql
sql.execute('SELECT * FROM table_name', engine)
sql.execute('INSERT INTO table_name VALUES(?, ?, ?)', engine,
            params=[('id', 1, 12.2, True)])
```

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.

Scroll To Top

```
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 [541]: import sqlalchemy as sa

In [542]: pd.read_sql(sa.text('SELECT * FROM data where Col_1=:col1'),
   .....:             engine, params={'col1': 'X'})
   .....:
Out[542]:
   index  id                        Date Col_1  Col_2  Col_3
0      0  26  2010-10-18 00:00:00.000000     X   27.5      1
```

If you have an SQLAlchemy description of your database you can express where conditions using SQLAlchemy expressions

```
In [543]: metadata = sa.MetaData()

In [544]: 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 [545]: pd.read_sql(sa.select([data_table]).where(data_table.c.Col_3 is True), engine
Out[545]:
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()**

Scroll To Top

```
In [546]: import datetime as dt

In [547]: expr = sa.select([data_table]).where(data_table.c.Date > sa.bindparam('date'))

In [548]: pd.read_sql(expr, engine, params={'date': dt.datetime(2010, 10, 18)})
Out[548]: 
   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:')
```

And then issue the following queries:

```
data.to_sql('data', con)
pd.read_sql_query("SELECT * FROM data", con)
```

Google BigQuery

Warning: Starting in 0.20.0, pandas has split off Google BigQuery support into the separate package `pandas-gbq`. You can `pip install pandas-gbq` to get it.

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 **`to_stata()`** will write a DataFrame into a .dta file. The format version of this file is always 115 (Stata 12).

Scroll To Top

```
In [549]: df = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [550]: 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.

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 **`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 **`StataReader`** that can be used to read the file incrementally.

```
In [551]: pd.read_stata('stata.dta')
Out[551]: 
   index         A         B
0      0  1.031231  0.196447
1      1  0.190188  0.619078
2      2  0.036658 -0.100501
3      3  0.201772  1.763002
4      4  0.454977 -1.958922
5      5 -0.628529  0.133171
6      6 -1.274374  2.518925
7      7 -0.517547 -0.360773
8      8  0.877961 -1.881598
9      9 -0.699067 -1.566913
```

Specifying a `chunksize` yields a **`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.

Scroll To Top

```
In [552]: reader = pd.read_stata('stata.dta', chunksize=3)

In [553]: for df in reader:
   .....:     print(df.shape)
   .....:
(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 [554]: reader = pd.read_stata('stata.dta', iterator=True)

In [555]: chunk1 = reader.read(5)

In [556]: 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.

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.

Scroll To Top

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.

> **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.

SAS formats

The top-level function **`read_sas()`** can read (but not write) SAS *xport* (.XPT) and (since *v0.18.0*) *SAS7BDAT* (.sas7bdat) format files.

SAS files only contain two value types: ASCII text and floating point values (usually 8 bytes but sometimes truncated). For xport files, there is no automatic type conversion to integers, dates, or categoricals. For SAS7BDAT files, the format codes may allow date variables to be automatically converted to dates. By default the whole file is read and returned as a `DataFrame`.

Specify a `chunksize` or use `iterator=True` to obtain reader objects (`XportReader` or `SAS7BDATReader`) for incrementally reading the file. The reader objects also have attributes that contain additional information about the file and its variables.

Read a SAS7BDAT file:

```
df = pd.read_sas('sas_data.sas7bdat')
```

Obtain an iterator and read an XPORT file 100,000 lines at a time:

```
def do_something(chunk):
    pass

rdr = pd.read_sas('sas_xport.xpt', chunk=100000)
for chunk in rdr:
    do_something(chunk)
```

Scroll To Top

The specification for the xport file format is available from the SAS web site.

No official documentation is available for the SAS7BDAT format.

Other file formats

pandas itself only supports IO with a limited set of file formats that map cleanly to its tabular data model. For reading and writing other file formats into and from pandas, we recommend these packages from the broader community.

netCDF

xarray provides data structures inspired by the pandas `DataFrame` for working with multi-dimensional datasets, with a focus on the netCDF file format and easy conversion to and from pandas.

Performance considerations

This is an informal comparison of various IO methods, using pandas 0.20.3. Timings are machine dependent and small differences should be ignored.

```
In [1]: sz = 1000000
In [2]: df = pd.DataFrame({'A': np.random.randn(sz), 'B': [1] * sz})

In [3]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 2 columns):
A    1000000 non-null float64
B    1000000 non-null int64
dtypes: float64(1), int64(1)
memory usage: 15.3 MB
```

Given the next test set:

```
from numpy.random import randn

sz = 1000000
df = pd.DataFrame({'A': randn(sz), 'B': [1] * sz})


def test_sql_write(df):
    if os.path.exists('test.sql'):
        os.remove('test.sql')
    sql_db = sqlite3.connect('test.sql')
    df.to_sql(name='test_table', con=sql_db)
    sql_db.close()


def test_sql_read():
    sql_db = sqlite3.connect('test.sql')
    pd.read_sql_query("select * from test_table", sql_db)
    sql_db.close()
```

Scroll To Top

```
def test_hdf_fixed_write(df):
    df.to_hdf('test_fixed.hdf', 'test', mode='w')


def test_hdf_fixed_read():
    pd.read_hdf('test_fixed.hdf', 'test')


def test_hdf_fixed_write_compress(df):
    df.to_hdf('test_fixed_compress.hdf', 'test', mode='w', complib='blosc')


def test_hdf_fixed_read_compress():
    pd.read_hdf('test_fixed_compress.hdf', 'test')


def test_hdf_table_write(df):
    df.to_hdf('test_table.hdf', 'test', mode='w', format='table')


def test_hdf_table_read():
    pd.read_hdf('test_table.hdf', 'test')


def test_hdf_table_write_compress(df):
    df.to_hdf('test_table_compress.hdf', 'test', mode='w',
              complib='blosc', format='table')


def test_hdf_table_read_compress():
    pd.read_hdf('test_table_compress.hdf', 'test')


def test_csv_write(df):
    df.to_csv('test.csv', mode='w')


def test_csv_read():
    pd.read_csv('test.csv', index_col=0)


def test_feather_write(df):
    df.to_feather('test.feather')


def test_feather_read():
    pd.read_feather('test.feather')


def test_pickle_write(df):
    df.to_pickle('test.pkl')


def test_pickle_read():
    pd.read_pickle('test.pkl')


def test_pickle_write_compress(df):
    df.to_pickle('test.pkl.compress', compression='xz')


def test_pickle_read_compress():
    pd.read_pickle('test.pkl.compress', compression='xz')
```

Scroll To Top

When writing, the top-three functions in terms of speed are are `test_pickle_write`, `test_feather_write` and `test_hdf_fixed_write_compress`.

```
In [14]: %timeit test_sql_write(df)
2.37 s ± 36.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [15]: %timeit test_hdf_fixed_write(df)
194 ms ± 65.9 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [26]: %timeit test_hdf_fixed_write_compress(df)
119 ms ± 2.15 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [16]: %timeit test_hdf_table_write(df)
623 ms ± 125 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [27]: %timeit test_hdf_table_write_compress(df)
563 ms ± 23.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [17]: %timeit test_csv_write(df)
3.13 s ± 49.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [30]: %timeit test_feather_write(df)
103 ms ± 5.88 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [31]: %timeit test_pickle_write(df)
109 ms ± 3.72 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [32]: %timeit test_pickle_write_compress(df)
3.33 s ± 55.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

When reading, the top three are `test_feather_read`, `test_pickle_read` and `test_hdf_fixed_read`.

```
In [18]: %timeit test_sql_read()
1.35 s ± 14.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [19]: %timeit test_hdf_fixed_read()
14.3 ms ± 438 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [28]: %timeit test_hdf_fixed_read_compress()
23.5 ms ± 672 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [20]: %timeit test_hdf_table_read()
35.4 ms ± 314 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [29]: %timeit test_hdf_table_read_compress()
42.6 ms ± 2.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [22]: %timeit test_csv_read()
516 ms ± 27.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [33]: %timeit test_feather_read()
4.06 ms ± 115 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [34]: %timeit test_pickle_read()
6.5 ms ± 172 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [35]: %timeit test_pickle_read_compress()
588 ms ± 3.57 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Scroll To Top

Space on disk (in bytes)

```
34816000 Aug 21 18:00 test.sql
24009240 Aug 21 18:00 test_fixed.hdf
 7919610 Aug 21 18:00 test_fixed_compress.hdf
24458892 Aug 21 18:00 test_table.hdf
 8657116 Aug 21 18:00 test_table_compress.hdf
28520770 Aug 21 18:00 test.csv
16000248 Aug 21 18:00 test.feather
16000848 Aug 21 18:00 test.pkl
 7554108 Aug 21 18:00 test.pkl.compress
```

Scroll To Top