IO Tools (Text, CSV, HDF5, ...)

The pandas I/O API is a set of top level reader functions accessed like pd.read_csv() that generally return a pandas object. The corresponding writer functions are object methods that are accessed like df.to_csv()

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	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
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 two workhorse functions for reading text files (a.k.a. flat files) are read_csv() and read_table(). They both use the same parsing code to intelligently convert tabular data into a DataFrame object. See the cookbook for some advanced strategies.

Parsing options

read_csv() and read_table() accept the following 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).

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 automatically. 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 as if header=0 if no names passed, otherwise as if 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 multi-index 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 unless mangle_dupe_cols=True, which is the default.

index_col : *int or sequence or False, default None*

Column to use as the row labels of the DataFrame. If a sequence is given, a MultiIndex is used. If you have a malformed file with delimiters at the end of each line, you might consider index_col=False to force pandas to *not* use the first column as the index (row names).

usecols : *array-like or callable, default None*

Return a subset of the columns. If array-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 array-like *usecols* parameter would be [0, 1, 2] or [‘foo’, ‘bar’, ‘baz’].

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

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

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

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

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

as_recarray : *boolean, default False*

DEPRECATED: this argument will be removed in a future version. Please call pd.read_csv(...).to_records() instead.

Return a NumPy recarray instead of a DataFrame after parsing the data. If set to True, this option takes precedence over the squeeze parameter. In addition, as row indices are not available in such a format, the index_col parameter will be ignored.

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.0'...'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* 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.

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

In [5]: pd.read_csv(StringIO(data))
Out[5]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3
```

```
In [6]: pd.read_csv(StringIO(data), skiprows=lambda x: x % 2 != 0)
Out[6]:
  col1 col2  col3
0    a    b     2
```

skipfooter : *int, default 0*

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

skip_footer : *int, default 0*

DEPRECATED: use the skipfooter parameter instead, as they are identical

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)

buffer_lines : *int, default None*

DEPRECATED: this argument will be removed in a future version because its value is not respected by the parser

compact_ints : *boolean, default False*

DEPRECATED: this argument will be removed in a future version

If compact_ints is True, then for any column that is of integer dtype, the parser will attempt to cast it as the smallest integer dtype possible, either signed or unsigned depending on the specification from the use_unsigned parameter.

use_unsigned : *boolean, default False*

DEPRECATED: this argument will be removed in a future version

If integer columns are being compacted (i.e. compact_ints=True), specify whether the column should be compacted to the smallest signed or unsigned integer dtype.

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

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

keep_default_na : *boolean, default True*
: If na_values are specified and keep_default_na is `False` the default NaN values are overridden, otherwise they're appended to.

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.

dayfirst : *boolean, default False*

DD/MM format dates, international and European format.

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.

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).

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.

tupleize_cols : *boolean, default False*

Leave a list of tuples on columns as is (default is to convert to a MultiIndex on the columns).

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

Starting with v0.10, you can indicate the data type for the whole DataFrame or individual columns:

```
In [7]: data = 'a,b,c\n1,2,3\n4,5,6\n7,8,9'

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

In [9]: df = pd.read_csv(StringIO(data), dtype=object)

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

In [11]: df['a'][0]
Out[11]: '1'

In [12]: df = pd.read_csv(StringIO(data), dtype={'b': object, 'c': np.float64})

In [13]: df.dtypes
Out[13]:
a      int64
b     object
c    float64
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 [14]: data = "col_1\n1\n2\n'A'\n4.22"

In [15]: df = pd.read_csv(StringIO(data), converters={'col_1':str})
```

```
In [16]: df
Out[16]:
  col_1
0     1
1     2
2   'A'
3  4.22

In [17]: df['col_1'].apply(type).value_counts()
Out[17]:
<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 [18]: df2 = pd.read_csv(StringIO(data))

In [19]: df2['col_1'] = pd.to_numeric(df2['col_1'], errors='coerce')

In [20]: df2
Out[20]:
   col_1
0   1.00
1   2.00
2    NaN
3   4.22

In [21]: df2['col_1'].apply(type).value_counts()
Out[21]:
<class 'float'>    4
Name: col_1, dtype: int64
```

which would 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 [22]: df = pd.DataFrame({'col_1': list(range(500000)) + ['a', 'b'] + list(range(500000

In [23]: df.to_csv('foo.csv')

In [24]: mixed_df = pd.read_csv('foo.csv')

In [25]: mixed_df['col_1'].apply(type).value_counts()
Out[25]: 
<class 'int'>    737858
<class 'str'>    262144
Name: col_1, dtype: int64

In [26]: mixed_df['col_1'].dtype
Out[26]: 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'

```
In [27]: data = 'col1,col2,col3\na,b,1\na,b,2\nc,d,3'

In [28]: pd.read_csv(StringIO(data))
Out[28]: 
  col1 col2  col3
0    a    b     1
1    a    b     2
```

```
2   c   d    3

In [29]: pd.read_csv(StringIO(data)).dtypes
Out[29]:
col1    object
col2    object
col3     int64
dtype: object

In [30]: pd.read_csv(StringIO(data), dtype='category').dtypes
Out[30]:
col1    category
col2    category
col3    category
dtype: object
```

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

```
In [31]: pd.read_csv(StringIO(data), dtype={'col1': 'category'}).dtypes
Out[31]:
col1    category
col2      object
col3       int64
dtype: object
```

Note: 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().

```
In [32]: df = pd.read_csv(StringIO(data), dtype='category')

In [33]: df.dtypes
Out[33]:
col1    category
col2    category
col3    category
dtype: object

In [34]: df['col3']
Out[34]:
0    1
```

```
1    2
2    3
Name: col3, dtype: category
Categories (3, object): [1, 2, 3]

In [35]: df['col3'].cat.categories = pd.to_numeric(df['col3'].cat.categories)

In [36]: df['col3']
Out[36]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, int64): [1, 2, 3]
```

Naming and Using Columns

Handling column names

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

```
In [37]: data = 'a,b,c\n1,2,3\n4,5,6\n7,8,9'

In [38]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [39]: pd.read_csv(StringIO(data))
Out[39]:
   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 [40]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [41]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
Out[41]:
   foo  bar  baz
0    1    2    3
1    4    5    6
2    7    8    9

In [42]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)
Out[42]:
  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 [43]: data = 'skip this skip it\na,b,c\n1,2,3\n4,5,6\n7,8,9'

In [44]: pd.read_csv(StringIO(data), header=1)
Out[44]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

Duplicate names parsing

If the file or header contains duplicate names, pandas by default will deduplicate these names so as to prevent data overwrite:

```
In [45]: data = 'a,b,a\n0,1,2\n3,4,5'
```

```
In [46]: pd.read_csv(StringIO(data))
Out[46]:
   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.0’...’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:

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

```
In [47]: data = 'a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz'

In [48]: pd.read_csv(StringIO(data))
Out[48]:
   a  b  c    d
0  1  2  3  foo
```

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

In [49]: pd.read_csv(StringIO(data), usecols=['b', 'd'])
Out[49]: 
   b    d
0  2  foo
1  5  bar
2  8  baz

In [50]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
Out[50]: 
   a  c    d
0  1  3  foo
1  4  6  bar
2  7  9  baz

In [51]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['A', 'C'])
Out[51]: 
   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 [52]: pd.read_csv(StringIO(data), usecols=lambda x: x not in ['a', 'c'])
Out[52]: 
   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. Both of these are API changes introduced in version 0.15.

```
In [53]: data = '\na,b,c\n  \n# commented line\n1,2,3\n\n4,5,6'

In [54]: print(data)

a,b,c

# commented line
1,2,3

4,5,6

In [55]: pd.read_csv(StringIO(data), comment='#')
Out[55]: 
   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 [56]: data = 'a,b,c\n\n1,2,3\n\n\n4,5,6'

In [57]: pd.read_csv(StringIO(data), skip_blank_lines=False)
Out[57]: 
     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 [58]: data = '#comment\na,b,c\nA,B,C\n1,2,3'

In [59]: pd.read_csv(StringIO(data), comment='#', header=1)
Out[59]: 
```

```
   A  B  C
0  1  2  3

In [60]: data = 'A,B,C\n#comment\na,b,c\n1,2,3'

In [61]: pd.read_csv(StringIO(data), comment='#', skiprows=2)
Out[61]:
   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 [62]: data = '# empty\n# second empty line\n# third empty' \

In [62]: 'line\nX,Y,Z\n1,2,3\nA,B,C\n1,2.,4.\n5.,NaN,10.0'

In [63]: 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 [64]: pd.read_csv(StringIO(data), comment='#', skiprows=4, header=1)
Out[64]:
     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 [65]: 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 [66]: df = pd.read_csv('tmp.csv')

In [67]: df
Out[67]: 
         ID    level                        category
0  Patient1   123000           x # really unpleasant
1  Patient2    23000  y # wouldn't take his medicine
2  Patient3  1234018                     z # awesome
```

We can suppress the comments using the comment keyword:

```
In [68]: df = pd.read_csv('tmp.csv', comment='#')

In [69]: df
Out[69]: 
         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 [70]: data = b'word,length\nTr\xc3\xa4umen,7\nGr\xc3\xbc\xc3\x9fe,5'.decode('utf

In [71]: df = pd.read_csv(BytesIO(data), encoding='latin-1')

In [72]: df
Out[72]: 
      word  length
0  Träumen       7
```

```
1   Grüße      5

In [73]: df['word'][1]
Out[73]: '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 [74]: data = 'a,b,c\n4,apple,bat,5.7\n8,orange,cow,10'

In [75]: pd.read_csv(StringIO(data))
Out[75]: 
        a    b     c
4   apple  bat   5.7
8  orange  cow  10.0
```

```
In [76]: data = 'index,a,b,c\n4,apple,bat,5.7\n8,orange,cow,10'

In [77]: pd.read_csv(StringIO(data), index_col=0)
Out[77]: 
            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 [78]: data = 'a,b,c\n4,apple,bat,\n8,orange,cow,'

In [79]: print(data)
```

```
a,b,c
4,apple,bat,
8,orange,cow,

In [80]: pd.read_csv(StringIO(data))
Out[80]:
        a    b   c
4   apple  bat NaN
8  orange  cow NaN

In [81]: pd.read_csv(StringIO(data), index_col=False)
Out[81]:
   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 [82]: data = 'a,b,c\n4,apple,bat,\n8,orange,cow,'

In [83]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [84]: pd.read_csv(StringIO(data), usecols=['b', 'c'])
Out[84]:
     b   c
4  bat NaN
8  cow NaN

In [85]: pd.read_csv(StringIO(data), usecols=['b', 'c'], index_col=0)
Out[85]:
     b   c
4  bat NaN
8  cow NaN
```

Date Handling

Specifying Date Columns

To better facilitate working with datetime data, read_csv() and read_table() use 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 [86]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)

In [87]: df
Out[87]:
           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 [88]: df.index
Out[88]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype='datetime64[
```

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 [89]: 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 [90]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])

In [91]: df
Out[91]:
                  1_2                 1_3     0     4
```

```
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

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

```
In [92]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]],
   ....:                  keep_date_col=True)
   ....: 

In [93]: df
Out[93]: 
                  1_2                 1_3     0         1          2  \
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  19990127   19:00:00   
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  19990127   20:00:00   
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD  19990127   21:00:00   
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD  19990127   21:00:00   
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD  19990127   22:00:00   
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD  19990127   23:00:00   

          3     4  
0  18:56:00  0.81  
1  19:56:00  0.01  
2  20:56:00 -0.59  
3  21:18:00 -0.99  
4  21:56:00 -0.59  
5  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 [94]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}
```

```
In [95]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)

In [96]: df
Out[96]:
              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 [97]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [98]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   ....:                  index_col=0) #index is the nominal column
   ....:

In [99]: df
Out[99]:
                                 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 unparseable 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 [100]: import pandas.io.date_converters as conv

In [101]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   .....:                  date_parser=conv.parse_date_time)
   .....:

In [102]: df
Out[102]: 
              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']))
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.

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"

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 [103]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
   .....:                  infer_datetime_format=True)
   .....:
```

```
In [104]: df
Out[104]:
           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 [105]: print(open('tmp.csv').read())
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c

In [106]: pd.read_csv('tmp.csv', parse_dates=[0])
Out[106]:
        date  value cat
0 2000-01-06      5   a
1 2000-02-06     10   b
2 2000-03-06     15   c

In [107]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])
Out[107]:
        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 [108]: val = '0.3066101993807095471566981359501369297504425048828125'

In [109]: data = 'a,b,c\n1,2,{0}'.format(val)

In [110]: abs(pd.read_csv(StringIO(data), engine='c', float_precision=None)['c'][0] - floa
Out[110]: 1.1102230246251565e-16

In [111]: abs(pd.read_csv(StringIO(data), engine='c', float_precision='high')['c'][0] - floa
Out[111]: 5.551115123125783e-17

In [112]: abs(pd.read_csv(StringIO(data), engine='c', float_precision='round_trip')['c'][0
Out[112]: 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 [113]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [114]: df = pd.read_csv('tmp.csv', sep='|')

In [115]: df
Out[115]: 
         ID      level category
0  Patient1    123,000        x
1  Patient2     23,000        y
2  Patient3  1,234,018        z

In [116]: df.level.dtype
Out[116]: dtype('O')
```

The thousands keyword allows integers to be parsed correctly

```
In [117]: print(open('tmp.csv').read())
```

```
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [118]: df = pd.read_csv('tmp.csv', sep='|', thousands=',')

In [119]: df
Out[119]: 
         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z

In [120]: df.level.dtype
Out[120]: dtype('int64')
```

NA Values

To control which values are parsed as missing values (which are signified by NaN), specifiy 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', 'NA', '#NA', 'NULL', 'NaN', '-NaN', 'nan', '-nan']. Although a 0-length string '' is not included in the default NaN values list, it is still treated as a missing value.

```
read_csv(path, na_values=[5])
```

the default values, in addition to 5 , 5.0 when interpreted as numbers are recognized as NaN

```
read_csv(path, keep_default_na=False, na_values=[""])
```

only an empty field will be NaN

```
read_csv(path, keep_default_na=False, na_values=["NA", "0"])
```

only NA and 0 as strings are NaN

```
read_csv(path, 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 [121]: print(open('tmp.csv').read())
level
Patient1,123000
Patient2,23000
Patient3,1234018

In [122]: output =  pd.read_csv('tmp.csv', squeeze=True)

In [123]: output
Out[123]:
Patient1     123000
Patient2      23000
Patient3    1234018
Name: level, dtype: int64

In [124]: type(output)
Out[124]: pandas.core.series.Series
```

Boolean values

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

```
In [125]: data= 'a,b,c\n1,Yes,2\n3,No,4'

In [126]: print(data)
a,b,c
1,Yes,2
3,No,4

In [127]: pd.read_csv(StringIO(data))
Out[127]:
   a    b  c
0  1  Yes  2
1  3   No  4

In [128]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
Out[128]:
   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 will cause an error by default:

```
In [27]: data = 'a,b,c\n1,2,3\n4,5,6,7\n8,9,10'

In [28]: pd.read_csv(StringIO(data))
---------------------------------------------------------------------------
ParserError                               Traceback (most recent call last)
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 [129]: 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 [130]: dia = csv.excel()

In [131]: dia.quoting = csv.QUOTE_NONE

In [132]: pd.read_csv(StringIO(data), dialect=dia)
Out[132]:
       label1 label2 label3
```

```
index1    "a    c    e
index2     b    d    f
```

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

```
In [133]: data = 'a,b,c~1,2,3~4,5,6'

In [134]: pd.read_csv(StringIO(data), lineterminator='~')
Out[134]:
   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 [135]: data = 'a, b, c\n1, 2, 3\n4, 5, 6'

In [136]: print(data)
a, b, c
1, 2, 3
4, 5, 6

In [137]: pd.read_csv(StringIO(data), skipinitialspace=True)
Out[137]:
   a  b  c
0  1  2  3
1  4  5  6
```

The parsers make every attempt to "do the right thing" and not be very fragile. Type inference is a pretty big deal. So if a column can be coerced to integer dtype without altering the contents, it 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 [138]: data = 'a,b\n"hello, \\"Bob\\", nice to see you",5'

In [139]: print(data)
a,b
"hello, \"Bob\", nice to see you",5

In [140]: pd.read_csv(StringIO(data), escapechar='\\')
Out[140]:
                               a  b
0  hello, "Bob", nice to see you  5
```

Files with Fixed Width Columns

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

- 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 behaviour, if not specified, is to infer.
- widths: A list of field widths which can be used instead of 'colspecs' if the intervals are contiguous.

Consider a typical fixed-width data file:

```
In [141]: 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 [142]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]
```

```
In [143]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)

In [144]: df
Out[144]:
                 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 [145]: widths = [6, 14, 13, 10]

In [146]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [147]: df
Out[147]:
        0           1           2        3
0  id8141  360.242940  149.910199  11950.7
1  id1594  444.953632  166.985655  11788.4
2  id1849  364.136849  183.628767  11806.2
3  id1230  413.836124  184.375703  11916.8
4  id1948  502.953953  173.237159  12468.3
```

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.

New in version 0.13.0.

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

```
In [149]: df
Out[149]:
                 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 [150]: pd.read_fwf('bar.csv', header=None, index_col=0).dtypes
Out[150]:
1    float64
2    float64
3    float64
dtype: object

In [151]: pd.read_fwf('bar.csv', header=None, dtype={2: 'object'}).dtypes
Out[151]:
0     object
1    float64
2     object
3    float64
dtype: object
```

Indexes

Files with an “implicit” index column

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

```
In [152]: 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 [153]: pd.read_csv('foo.csv')
Out[153]:
          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 [154]: df = pd.read_csv('foo.csv', parse_dates=True)

In [155]: df.index
Out[155]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype='datetime64
```

Reading an index with a MultiIndex

Suppose you have data indexed by two columns:

```
In [156]: 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 and read_table can take a list of column numbers to turn multiple columns into a MultiIndex for the index of the returned object:

```
In [157]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0,1])

In [158]: df
Out[158]:
            zit   xit
year indiv
1977 A      1.20  0.60
     B      1.50  0.50
     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 [159]: df.loc[1978]
Out[159]:
       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 order to have the pre-0.13 behavior of tupleizing columns, specify tupleize_cols=True.

```
In [160]: from pandas.util.testing import makeCustomDataframe as mkdf

In [161]: df = mkdf(5,3,r_idx_nlevels=2,c_idx_nlevels=4)

In [162]: df.to_csv('mi.csv')

In [163]: 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 [164]: pd.read_csv('mi.csv',header=[0,1,2,3],index_col=[0,1])
Out[164]: 
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
```

Starting in 0.13.0, read_csv will be able to interpret a more common format of multi-columns indices.

```
In [165]: 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 [166]: pd.read_csv('mi2.csv',header=[0,1],index_col=0)
Out[166]: 
```

```
    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 [167]: print(open('tmp2.sv').read())
:0:1:2:3
0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.13563237101
1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-1.044235966
2:-0.8618489633477999:-2.1045692188948086:-0.4949292740687813:1.07180380703
3:0.7215551622443669:-0.7067711336300845:-1.0395749851146963:0.27185988554
4:-0.4249723297888375З:0.567020349793672:0.27623201927771873:-1.0874006912
5:-0.6736897080883706:0.1136484096888855:-1.4784265524372235:0.52498766711
6:0.4047052186802365:0.5770459859204836:-1.7150020161146375:-1.03926848351
7:-0.3706468582364464:-1.1578922506419993:-1.344311812731667:0.844885141424
8:1.0757697837155533:-0.10904997528022223:1.6435630703622064:-1.4693879595
9:0.35702056413309086:-0.6746001037299882:-1.776903716971867:-0.96891381244


In [168]: pd.read_csv('tmp2.sv', sep=None, engine='python')
Out[168]:
   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

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 [169]: print(open('tmp.sv').read())
|0|1|2|3
0|0.4691122999071863|-0.2828633443286633|-1.5090585031735124|-1.135632371
1|1.2121120250208506|-0.17321464905330858|0.11920871129693428|-1.0442359
2|-0.8618489633477999|-2.1045692188948086|-0.4949292740687813|1.071803807
3|0.7215551622443669|-0.7067711336300845|-1.0395749851146963|0.271859885
4|-0.42497232978883753|0.567020349793672|0.27623201927771873|-1.08740069
5|-0.6736897080883706|0.1136484096888855|-1.4784265524372235|0.524987667
6|0.4047052186802365|0.5770459859204836|-1.7150020161146375|-1.039268483
7|-0.3706468582364464|-1.1578922506419993|-1.344311812731667|0.8448851414
8|1.0757697837155533|-0.10904997528022223|1.6435630703622064|-1.46938795
9|0.35702056413309086|-0.6746001037299882|-1.776903716971867|-0.968913812


In [170]: table = pd.read_table('tmp.sv', sep='|')

In [171]: table
Out[171]: 
   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 or read_table, the return value will be an iterable object of type TextFileReader:

```
In [172]: reader = pd.read_table('tmp.sv', sep='|', chunksize=4)

In [173]: reader
Out[173]: <pandas.io.parsers.TextFileReader at 0x128608160>

In [174]: 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
```

Specifying iterator=True will also return the TextFileReader object:

```
In [175]: reader = pd.read_table('tmp.sv', sep='|', iterator=True)

In [176]: reader.get_chunk(5)
Out[176]:
   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:

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

Writing out Data

Writing to CSV format

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

- path_or_buf: A string path to the file to write or a StringIO
- 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
- cols: 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 '\n')
- 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
- tupleize_cols: If False (default), write as a list of tuples, otherwise write in an expanded line format suitable for read_csv
- date_format: Format string for datetime objects

Writing a formatted string

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

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

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

JSON

Read and write JSON format files and strings.

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}

  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.

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

In [178]: json = dfj.to_json()

In [179]: json
Out[179]: '{"A":{"0":-1.2945235903,"1":0.2766617129,"2":-0.0139597524,"3":-0.006153
```

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

In [181]: dfjo
Out[181]: 
   A  B  C
x  1  4  7
y  2  5  8
z  3  6  9

In [182]: sjo = pd.Series(dict(x=15, y=16, z=17), name='D')

In [183]: sjo
Out[183]: 
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 [184]: dfjo.to_json(orient="columns")
Out[184]: '{"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 [185]: dfjo.to_json(orient="index")
Out[185]: '{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":{"A":3,"B":6,"C":9}}'

In [186]: sjo.to_json(orient="index")
Out[186]: '{"x":15,"y":16,"z":17}'
```

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

```
In [187]: dfjo.to_json(orient="records")
Out[187]: '[{"A":1,"B":4,"C":7},{"A":2,"B":5,"C":8},{"A":3,"B":6,"C":9}]'

In [188]: sjo.to_json(orient="records")
Out[188]: '[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 [189]: dfjo.to_json(orient="values")
Out[189]: '[[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 [190]: dfjo.to_json(orient="split")
Out[190]: '{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,5,8],[3,6,9]]}'
```

```
In [191]: sjo.to_json(orient="split")
Out[191]: '{"name":"D","index":["x","y","z"],"data":[15,16,17]}'
```

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

In [193]: dfd['date'] = pd.Timestamp('20130101')

In [194]: dfd = dfd.sort_index(1, ascending=False)

In [195]: json = dfd.to_json(date_format='iso')

In [196]: json
Out[196]: '{"date":{"0":"2013-01-01T00:00:00.000Z","1":"2013-01-01T00:00:00.000Z","
```

Writing in ISO date format, with microseconds

```
In [197]: json = dfd.to_json(date_format='iso', date_unit='us')

In [198]: json
Out[198]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00:00.000
```

Epoch timestamps, in seconds

```
In [199]: json = dfd.to_json(date_format='epoch', date_unit='s')

In [200]: json
Out[200]: '{"date":{"0":1356998400,"1":1356998400,"2":1356998400,"3":1356998400,"
```

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

```
In [201]: dfj2 = dfj.copy()

In [202]: dfj2['date'] = pd.Timestamp('20130101')

In [203]: dfj2['ints'] = list(range(5))

In [204]: dfj2['bools'] = True

In [205]: dfj2.index = pd.date_range('20130101', periods=5)

In [206]: dfj2.to_json('test.json')

In [207]: open('test.json').read()
Out[207]: '{"A":{"1356998400000":-1.2945235903,"1357084800000":0.2766617129,"13
```

Fallback Behavior

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

- if the dtype is unsupported (e.g. np.complex) then the default_handler, if provided, will be called for each value, otherwise an exception is raised.
- if an object is unsupported it will attempt the following:
  - check if the object has defined a toDict method and call it. A toDict method should return a dict which will then be JSON serialized.
  - invoke the default_handler if one was provided.
  - convert the object to a dict by traversing its contents. However this will often fail with an OverflowError or give unexpected results.

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

```
DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json()  # raises

RuntimeError: Unhandled numpy dtype 15
```

can be dealt with by specifying a simple default_handler:

```
In [208]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
```

```
Out[208]: '{"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}

  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

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

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

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

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

Data Conversion

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

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

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

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

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

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

Reading from a JSON string:

```
In [209]: pd.read_json(json)
Out[209]:
          A         B       date
0 -1.206412  2.565646 2013-01-01
1  1.431256  1.340309 2013-01-01
2 -1.170299 -0.226169 2013-01-01
3  0.410835  0.813850 2013-01-01
4  0.132003 -0.827317 2013-01-01
```

Reading from a file:

```
In [210]: pd.read_json('test.json')
Out[210]:
                   A         B  bools       date  ints
2013-01-01 -1.294524  0.413738   True 2013-01-01     0
2013-01-02  0.276662 -0.472035   True 2013-01-01     1
2013-01-03 -0.013960 -0.362543   True 2013-01-01     2
2013-01-04 -0.006154 -0.923061   True 2013-01-01     3
2013-01-05  0.895717  0.805244   True 2013-01-01     4
```

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

```
In [211]: pd.read_json('test.json', dtype=object).dtypes
Out[211]:
A        object
B        object
bools    object
date     object
```

```
ints     object
dtype: object
```

Specify dtypes for conversion:

```
In [212]: pd.read_json('test.json', dtype={'A' : 'float32', 'bools' : 'int8'}).dtypes
Out[212]: 
A                float32
B                float64
bools               int8
date      datetime64[ns]
ints               int64
dtype: object
```

Preserve string indices:

```
In [213]: si = pd.DataFrame(np.zeros((4, 4)),
   .....:          columns=list(range(4)),
   .....:          index=[str(i) for i in range(4)])
   .....: 

In [214]: si
Out[214]: 
     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 [215]: si.index
Out[215]: Index(['0', '1', '2', '3'], dtype='object')

In [216]: si.columns
Out[216]: Int64Index([0, 1, 2, 3], dtype='int64')

In [217]: json = si.to_json()

In [218]: sij = pd.read_json(json, convert_axes=False)

In [219]: sij
Out[219]: 
   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 [220]: sij.index
Out[220]: Index(['0', '1', '2', '3'], dtype='object')

In [221]: sij.columns
Out[221]: Index(['0', '1', '2', '3'], dtype='object')
```

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

```
In [222]: json = dfj2.to_json(date_unit='ns')

# Try to parse timestamps as millseconds -> Won't Work
In [223]: dfju = pd.read_json(json, date_unit='ms')

In [224]: dfju
Out[224]: 
                            A         B  bools                 date  ints
1356998400000000000 -1.294524  0.413738   True  1356998400000000000     0
1357084800000000000  0.276662 -0.472035   True  1356998400000000000     1
1357171200000000000 -0.013960 -0.362543   True  1356998400000000000     2
1357257600000000000 -0.006154 -0.923061   True  1356998400000000000     3
1357344000000000000  0.895717  0.805244   True  1356998400000000000     4

# Let pandas detect the correct precision
In [225]: dfju = pd.read_json(json)

In [226]: dfju
Out[226]: 
                   A         B  bools       date  ints
2013-01-01 -1.294524  0.413738   True 2013-01-01     0
2013-01-02  0.276662 -0.472035   True 2013-01-01     1
2013-01-03 -0.013960 -0.362543   True 2013-01-01     2
2013-01-04 -0.006154 -0.923061   True 2013-01-01     3
2013-01-05  0.895717  0.805244   True 2013-01-01     4

# Or specify that all timestamps are in nanoseconds
In [227]: dfju = pd.read_json(json, date_unit='ns')

In [228]: dfju
Out[228]: 
```

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

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

In [230]: randfloats.shape = (1000, 10)

In [231]: dffloats = pd.DataFrame(randfloats, columns=list('ABCDEFGHIJ'))

In [232]: jsonfloats = dffloats.to_json()
```

```
In [233]: timeit pd.read_json(jsonfloats)
7.19 ms +- 226 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [234]: timeit pd.read_json(jsonfloats, numpy=True)
4.86 ms +- 160 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

The speedup is less noticeable for smaller datasets:

```
In [235]: jsonfloats = dffloats.head(100).to_json()
```

```
In [236]: timeit pd.read_json(jsonfloats)
```

```
4.11 ms +- 177 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [237]: timeit pd.read_json(jsonfloats, numpy=True)
3.25 ms +- 136 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

New in version 0.13.0.

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

```
In [238]: from pandas.io.json import json_normalize

In [239]: 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 [240]: json_normalize(data, 'counties', ['state', 'shortname', ['info', 'governor']])
Out[240]:
         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
```

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.

```
In [241]: jsonl = '''
   .....:     {"a":1,"b":2}
   .....:     {"a":3,"b":4}
   .....: '''
   .....:

In [242]: df = pd.read_json(jsonl, lines=True)

In [243]: df
Out[243]:
   a  b
0  1  2
1  3  4

In [244]: df.to_json(orient='records', lines=True)
Out[244]: '{"a":1,"b":2}\n{"a":3,"b":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 [245]: 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 [246]: df
Out[246]: 
     A  B          C
idx                 
0    1  a 2016-01-01
1    2  b 2016-01-02
2    3  c 2016-01-03

In [247]: df.to_json(orient='table', date_format="iso")
Out[247]: '{"schema": {"fields":[{"name":"idx","type":"integer"},{"name":"A","type":"int
```

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

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

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

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

A few notes on the generated table schema:

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

```
In [248]: from pandas.io.json import build_table_schema

In [249]: s = pd.Series(pd.date_range('2016', periods=4))

In [250]: build_table_schema(s)
Out[250]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'datetime'}],
 'pandas_version': '0.20.0',
 'primaryKey': ['index']}
```

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

```
In [251]: s_tz = pd.Series(pd.date_range('2016', periods=12,
   .....:                                tz='US/Central'))
   .....:

In [252]: build_table_schema(s_tz)
Out[252]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'name': 'values', 'type': 'datetime', 'tz': 'US/Central'}],
 'pandas_version': '0.20.0',
 'primaryKey': ['index']}
```

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

In [254]: build_table_schema(s_per)
```

```
Out[254]:
{'fields': [{'freq': 'A-DEC', 'name': 'index', 'type': 'datetime'},
  {'name': 'values', 'type': 'integer'}],
 'pandas_version': '0.20.0',
 'primaryKey': ['index']}
```

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

```
In [255]: s_cat = pd.Series(pd.Categorical(['a', 'b', 'a']))

In [256]: build_table_schema(s_cat)
Out[256]:
{'fields': [{'name': 'index', 'type': 'integer'},
  {'constraints': {'enum': ['a', 'b']},
   'name': 'values',
   'ordered': False,
   'type': 'any'}],
 'pandas_version': '0.20.0',
 'primaryKey': ['index']}
```

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

```
In [257]: s_dupe = pd.Series([1, 2], index=[1, 1])

In [258]: build_table_schema(s_dupe)
Out[258]:
{'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 [259]: s_multi = pd.Series(1, index=pd.MultiIndex.from_product([('a', 'b'),
   .....:                                                        (0, 1)]))
   .....:

In [260]: build_table_schema(s_multi)
Out[260]:
```

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

- 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.

_Table Schema: http://specs.frictionlessdata.io/json-table-schema/

HTML

Reading HTML Content

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

New in version 0.12.0.

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 [261]: url = 'http://www.fdic.gov/bank/individual/failed/banklist.html'

In [262]: dfs = pd.read_html(url)

In [263]: dfs
Out[263]:
```

```
[                                        Bank Name                City  \
0                               Fayette County Bank          Saint Elmo
1    Guaranty Bank, (d/b/a BestBank in Georgia & Mi...           Milwaukee
2                                    First NBC Bank         New Orleans
3                                     Proficio Bank  Cottonwood Heights
4                     Seaway Bank and Trust Company             Chicago
5                            Harvest Community Bank          Pennsville
6                                       Allied Bank            Mulberry
..                                              ...                 ...
546                      Hamilton Bank, NA  En Espanol               Miami
547                          Sinclair National Bank            Gravette
548                              Superior Bank, FSB            Hinsdale
549                             Malta National Bank               Malta
550                   First Alliance Bank & Trust Co.          Manchester
551                 National State Bank of Metropolis          Metropolis
552                                Bank of Honolulu            Honolulu

     ST   CERT                Acquiring Institution        Closing Date  \
0    IL   1802           United Fidelity Bank, fsb        May 26, 2017
1    WI  30003  First-Citizens Bank & Trust Company         May 5, 2017
2    LA  58302                         Whitney Bank      April 28, 2017
3    UT  35495                    Cache Valley Bank       March 3, 2017
4    IL  19328                   State Bank of Texas    January 27, 2017
5    NJ  34951  First-Citizens Bank & Trust Company    January 13, 2017
6    AR     91                          Today's Bank  September 23, 2016
..   ..    ...                                  ...                 ...
546  FL  24382     Israel Discount Bank of New York    January 11, 2002
547  AR  34248                   Delta Trust & Bank   September 7, 2001
548  IL  32646                Superior Federal, FSB       July 27, 2001
549  OH   6629                    North Valley Bank         May 3, 2001
550  NH  34264  Southern New Hampshire Bank & Trust    February 2, 2001
551  IL   3815             Banterra Bank of Marion   December 14, 2000
552  HI  21029                   Bank of the Orient    October 13, 2000

           Updated Date
0         June 27, 2017
1          June 1, 2017
2          May 23, 2017
3          May 18, 2017
4          May 18, 2017
5          May 18, 2017
6     November 17, 2016
..                  ...
546  September 21, 2015
547   February 10, 2004
548     August 19, 2014
```

```
549   November 18, 2002
550   February 18, 2003
551      March 17, 2005
552      March 17, 2005

[553 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 [264]: with open(file_path, 'r') as f:
   .....:     dfs = pd.read_html(f.read())
   .....:

In [265]: dfs
Out[265]:
[                             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
 5                          Parkway Bank        Lenoir  NC  57158
 6              Chipola Community Bank      Marianna  FL  58034
 ..                                  ...           ...  ..    ...
 499              Hamilton Bank, NAEn Espanol         Miami  FL  24382
 500                Sinclair National Bank      Gravette  AR  34248
 501                    Superior Bank, FSB      Hinsdale  IL  32646
 502                   Malta National Bank         Malta  OH   6629
 503         First Alliance Bank & Trust Co.    Manchester  NH  34264
 504       National State Bank of Metropolis    Metropolis  IL   3815
 505                      Bank of Honolulu      Honolulu  HI  21029

                Acquiring Institution       Closing Date       Updated Date
 0                North Shore Bank, FSB       May 31, 2013       May 31, 2013
 1                   Western State Bank       May 14, 2013       May 20, 2013
 2                        Synovus Bank       May 10, 2013       May 21, 2013
 3                   Capital Bank, N.A.       May 10, 2013       May 14, 2013
 4                  Hamilton State Bank     April 26, 2013       May 16, 2013
 5       CertusBank, National Association     April 26, 2013       May 17, 2013
 6           First Federal Bank of Florida     April 19, 2013       May 16, 2013
```

```
..                                ...            ...            ...
499    Israel Discount Bank of New York   January 11, 2002      June 5, 2012
500                 Delta Trust & Bank  September 7, 2001  February 10, 2004
501               Superior Federal, FSB      July 27, 2001      June 5, 2012
502                  North Valley Bank        May 3, 2001  November 18, 2002
503  Southern New Hampshire Bank & Trust   February 2, 2001  February 18, 2003
504             Banterra Bank of Marion  December 14, 2000     March 17, 2005
505                 Bank of the Orient   October 13, 2000     March 17, 2005

[506 rows x 7 columns]]
```

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

```
In [266]: with open(file_path, 'r') as f:
   .....:     sio = StringIO(f.read())
   .....:

In [267]: dfs = pd.read_html(sio)

In [268]: dfs
Out[268]:
[                              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
 5                             Parkway Bank        Lenoir  NC  57158
 6                   Chipola Community Bank      Marianna  FL  58034
 ..                                     ...           ...  ..    ...
 499              Hamilton Bank, NAEn Espanol         Miami  FL  24382
 500                  Sinclair National Bank      Gravette  AR  34248
 501                      Superior Bank, FSB      Hinsdale  IL  32646
 502                     Malta National Bank         Malta  OH   6629
 503         First Alliance Bank & Trust Co.    Manchester  NH  34264
 504       National State Bank of Metropolis    Metropolis  IL   3815
 505                        Bank of Honolulu      Honolulu  HI  21029

                    Acquiring Institution       Closing Date       Updated Date
 0                   North Shore Bank, FSB       May 31, 2013       May 31, 2013
 1                     Western State Bank       May 14, 2013       May 20, 2013
 2                          Synovus Bank       May 10, 2013       May 21, 2013
 3                      Capital Bank, N.A.       May 10, 2013       May 14, 2013
 4                     Hamilton State Bank     April 26, 2013       May 16, 2013
```

```
5       CertusBank, National Association     April 26, 2013       May 17, 2013
6          First Federal Bank of Florida     April 19, 2013       May 16, 2013
..                                ...                ...               ...
499    Israel Discount Bank of New York   January 11, 2002        June 5, 2012
500                 Delta Trust & Bank  September 7, 2001   February 10, 2004
501                Superior Federal, FSB      July 27, 2001        June 5, 2012
502                    North Valley Bank        May 3, 2001   November 18, 2002
503  Southern New Hampshire Bank & Trust   February 2, 2001   February 18, 2003
504              Banterra Bank of Marion  December 14, 2000      March 17, 2005
505                   Bank of the Orient   October 13, 2000      March 17, 2005

[506 rows x 7 columns]]
```

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

Read a URL and match a table that contains specific text

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

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

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

Specify an index column

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

Specify a number of rows to skip

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

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

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

Specify an HTML attribute

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

Specify values that should be converted to NaN

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

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

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

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

or

```
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 [269]: df = pd.DataFrame(randn(2, 2))
```

```
In [270]: df
Out[270]: 
          0         1
0 -0.184744  0.496971
1 -0.856240  1.857977

In [271]: print(df.to_html())  # raw html
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

HTML:

	0	1
0	-0.184744	0.496971
1	-0.856240	1.857977

The columns argument will limit the columns shown

```
In [272]: 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 [273]: print(df.to_html(float_format='{0:.10f}'.format))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.1847438576</td>
      <td>0.4969711327</td>
    </tr>
    <tr>
```

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

HTML:

	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 [274]: 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	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.

```
In [275]: 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>
```

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

Escaped:

```
In [277]: 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 [278]: 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

- 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) and Excel 2007+ (.xlsx) files using the xlrd Python module. 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 sheetname indicating which sheet to parse.

```
# Returns a DataFrame
read_excel('path_to_file.xls', sheetname='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 be passed into read_excel There will be a performance benefit for reading multiple sheets as the file is read into memory only once.

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

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

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

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

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

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

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

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

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

New in version 0.12.

ExcelFile has been moved to the top level namespace.

New in version 0.17.

read_excel can take an ExcelFile object as input

Specifying Sheets

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

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

- The arguments sheetname allows specifying the sheet or sheets to read.
- The default value for sheetname 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
read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

Using the sheet index:

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

Using all default values:

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

Using None to get all sheets:

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

Using a list to get multiple sheets:

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

New in version 0.16.

read_excel can read more than one sheet, by setting sheetname to either a list of sheet names, a list of sheet positions, or None to read all sheets.

New in version 0.13.

Sheets can be specified by sheet index or sheet name, using an integer or string, respectively.

Reading a MultiIndex

New in version 0.17.

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

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

```
In [279]: df = pd.DataFrame({'a':[1,2,3,4], 'b':[5,6,7,8]},
   .....:                   index=pd.MultiIndex.from_product([['a','b'],['c','d']]))
   .....:

In [280]: df.to_excel('path_to_file.xlsx')

In [281]: df = pd.read_excel('path_to_file.xlsx', index_col=[0,1])

In [282]: df
Out[282]: 
     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 [283]: df.index = df.index.set_names(['lvl1', 'lvl2'])

In [284]: df.to_excel('path_to_file.xlsx')

In [285]: df = pd.read_excel('path_to_file.xlsx', index_col=[0,1])

In [286]: df
Out[286]: 
           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 [287]: df.columns = pd.MultiIndex.from_product([['a'],['b', 'd']], names=['c1', 'c2'])

In [288]: df.to_excel('path_to_file.xlsx')

In [289]: df = pd.read_excel('path_to_file.xlsx',
```

```
   .....:                  index_col=[0,1], header=[0,1])
   .....:

In [290]: df
Out[290]: 
c1         a  
c2         b  d
lvl1 lvl2      
a    c     1  5
     d     2  6
b    c     3  7
     d     4  8
```

Warning: Excel files saved in version 0.16.2 or prior that had index names will still able to be read in, but the has_index_names argument must specified to True.

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 *parse_cols* keyword to allow you to specify a subset of columns to parse.

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

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

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

```
read_excel('path_to_file.xls', 'Sheet1', parse_cols=[0, 2, 3])
```

Parsing Dates

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

```
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:

```
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:

```
cfun = lambda x: int(x) if x else -1
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.

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

Writing Excel Files

Writing Excel Files to Disk

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

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

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. One difference from 0.12.0 is that the index_label will be placed in the second row instead of the first. You can get the previous behaviour by setting the merge_cells option in to_excel() to False:

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

The Panel class also has a to_excel instance method, which writes each DataFrame in the Panel to a separate sheet.

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

```
with 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

New in version 0.17.

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

New in version 0.17.

Added support for Openpyxl >= 2.2

```
# 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 = 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.

Excel writer engines

New in version 0.13.

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 and openpyxl for .xlsm files 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: This includes stable support for Openpyxl from 1.6.1. However, it is advised to use version 2.2 and higher, especially when working with styles.
- xlsxwriter
- xlwt

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

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

# Or via pandas configuration.
from pandas import options
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)

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_table 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()
```

```
In [291]: clipdf
Out[291]: 
  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.

```
In [292]: df = pd.DataFrame(randn(5,3))

In [293]: df
Out[293]: 
          0         1         2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129

In [294]: df.to_clipboard()

In [295]: pd.read_clipboard()
Out[295]: 
          0         1         2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129
```

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 gtk or PyQt4 modules) 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 [296]: df
Out[296]:
          0         1         2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129

In [297]: 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 [298]: pd.read_pickle('foo.pkl')
Out[298]:
          0         1         2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129
```

Warning: Loading pickled data received from untrusted sources can be unsafe.

See: http://docs.python.org/2.7/library/pickle.html

Warning: Several internal refactorings, 0.13 (Series Refactoring), and 0.15 (Index Refactoring), preserve compatibility with pickles created prior to these versions. However, these must be read with pd.read_pickle, rather than the default python pickle.load. See this question for a detailed explanation.

Compressed pickle files

New in version 0.20.0.

read_pickle(), DataFame.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. *zip*` file supports read only and must contain only one data file to be read in.

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

In [300]: df
Out[300]: 
            A    B                   C
0    0.478412  foo 2013-01-01 00:00:00
1   -0.783748  foo 2013-01-01 00:00:01
2    1.403558  foo 2013-01-01 00:00:02
3   -0.539282  foo 2013-01-01 00:00:03
4   -1.651012  foo 2013-01-01 00:00:04
5    0.692072  foo 2013-01-01 00:00:05
6    1.022171  foo 2013-01-01 00:00:06
..        ...  ...                 ...
993 -1.613932  foo 2013-01-01 00:16:33
994  1.088104  foo 2013-01-01 00:16:34
995 -0.632963  foo 2013-01-01 00:16:35
996 -0.585314  foo 2013-01-01 00:16:36
997 -0.275038  foo 2013-01-01 00:16:37
998 -0.937512  foo 2013-01-01 00:16:38
999  0.632369  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Using an explicit compression type

```
In [301]: df.to_pickle("data.pkl.compress", compression="gzip")
```

```
In [302]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [303]: rt
Out[303]: 
            A    B                   C
0    0.478412  foo 2013-01-01 00:00:00
1   -0.783748  foo 2013-01-01 00:00:01
2    1.403558  foo 2013-01-01 00:00:02
3   -0.539282  foo 2013-01-01 00:00:03
4   -1.651012  foo 2013-01-01 00:00:04
5    0.692072  foo 2013-01-01 00:00:05
6    1.022171  foo 2013-01-01 00:00:06
..        ...  ...                 ...
993 -1.613932  foo 2013-01-01 00:16:33
994  1.088104  foo 2013-01-01 00:16:34
995 -0.632963  foo 2013-01-01 00:16:35
996 -0.585314  foo 2013-01-01 00:16:36
997 -0.275038  foo 2013-01-01 00:16:37
998 -0.937512  foo 2013-01-01 00:16:38
999  0.632369  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Inferring compression type from the extension

```
In [304]: df.to_pickle("data.pkl.xz", compression="infer")

In [305]: rt = pd.read_pickle("data.pkl.xz", compression="infer")

In [306]: rt
Out[306]: 
            A    B                   C
0    0.478412  foo 2013-01-01 00:00:00
1   -0.783748  foo 2013-01-01 00:00:01
2    1.403558  foo 2013-01-01 00:00:02
3   -0.539282  foo 2013-01-01 00:00:03
4   -1.651012  foo 2013-01-01 00:00:04
5    0.692072  foo 2013-01-01 00:00:05
6    1.022171  foo 2013-01-01 00:00:06
..        ...  ...                 ...
993 -1.613932  foo 2013-01-01 00:16:33
994  1.088104  foo 2013-01-01 00:16:34
995 -0.632963  foo 2013-01-01 00:16:35
996 -0.585314  foo 2013-01-01 00:16:36
```

```
997 -0.275038  foo 2013-01-01 00:16:37
998 -0.937512  foo 2013-01-01 00:16:38
999  0.632369  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

The default is to ‘infer

```
In [307]: df.to_pickle("data.pkl.gz")

In [308]: rt = pd.read_pickle("data.pkl.gz")

In [309]: rt
Out[309]: 
            A    B                   C
0    0.478412  foo 2013-01-01 00:00:00
1   -0.783748  foo 2013-01-01 00:00:01
2    1.403558  foo 2013-01-01 00:00:02
3   -0.539282  foo 2013-01-01 00:00:03
4   -1.651012  foo 2013-01-01 00:00:04
5    0.692072  foo 2013-01-01 00:00:05
6    1.022171  foo 2013-01-01 00:00:06
..        ...  ...                 ...
993 -1.613932  foo 2013-01-01 00:16:33
994  1.088104  foo 2013-01-01 00:16:34
995 -0.632963  foo 2013-01-01 00:16:35
996 -0.585314  foo 2013-01-01 00:16:36
997 -0.275038  foo 2013-01-01 00:16:37
998 -0.937512  foo 2013-01-01 00:16:38
999  0.632369  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]

In [310]: df["A"].to_pickle("s1.pkl.bz2")

In [311]: rt = pd.read_pickle("s1.pkl.bz2")

In [312]: rt
Out[312]: 
0      0.478412
1     -0.783748
2      1.403558
3     -0.539282
4     -1.651012
```

```
5      0.692072
6      1.022171
         ...
993   -1.613932
994    1.088104
995   -0.632963
996   -0.585314
997   -0.275038
998   -0.937512
999    0.632369
Name: A, Length: 1000, dtype: float64
```

msgpack

New in version 0.13.0.

Starting in 0.13.0, pandas is supporting 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: This is a very new feature of pandas. We intend to provide certain optimizations in the io of the msgpack data. Since this is marked as an EXPERIMENTAL LIBRARY, the storage format may not be stable until a future release.

As a result of writing format changes and other issues:

Packed with	Can be unpacked with
pre-0.17 / Python 2	any
pre-0.17 / Python 3	any
0.17 / Python 2	• 0.17 / Python 2 • >=0.18 / any Python
0.17 / Python 3	>=0.18 / any Python
0.18	>= 0.18

Reading (files packed by older versions) is backward-compatibile, except for files packed with 0.17 in Python 2, in which case only they can only be unpacked in Python 2.

```
In [313]: df = pd.DataFrame(np.random.rand(5,2),columns=list('AB'))
```

```
In [314]: df.to_msgpack('foo.msg')

In [315]: pd.read_msgpack('foo.msg')
Out[315]:
          A         B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359

In [316]: 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 [317]: pd.to_msgpack('foo.msg', df, 'foo', np.array([1,2,3]), s)

In [318]: pd.read_msgpack('foo.msg')
Out[318]:
[          A         B
 0  0.170801  0.895366
 1  0.838238  0.052592
 2  0.664140  0.289750
 3  0.449593  0.872087
 4  0.983618  0.744359, 'foo', array([1, 2, 3]), 2013-01-01    0.548134
 2013-01-02    0.503447
 2013-01-03    0.348438
 2013-01-04    0.707267
 2013-01-05    0.261656
 Freq: D, dtype: float64]
```

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

```
In [319]: for o in pd.read_msgpack('foo.msg',iterator=True):
   .....:     print o
   .....:
  File "<ipython-input-319-a0f40395739e>", line 2
    print o
          ^
SyntaxError: Missing parentheses in call to 'print'
```

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

```
In [320]: df.to_msgpack('foo.msg',append=True)

In [321]: pd.read_msgpack('foo.msg')
Out[321]: 
[          A         B
 0  0.170801  0.895366
 1  0.838238  0.052592
 2  0.664140  0.289750
 3  0.449593  0.872087
 4  0.983618  0.744359, 'foo', array([1, 2, 3]), 2013-01-01    0.548134
 2013-01-02    0.503447
 2013-01-03    0.348438
 2013-01-04    0.707267
 2013-01-05    0.261656
 Freq: D, dtype: float64,           A         B
 0  0.170801  0.895366
 1  0.838238  0.052592
 2  0.664140  0.289750
 3  0.449593  0.872087
 4  0.983618  0.744359]
```

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 [322]: pd.to_msgpack('foo2.msg', { 'dict' : [ { 'df' : df }, { 'string' : 'foo' }, { 'scalar' : 1.

In [323]: pd.read_msgpack('foo2.msg')
Out[323]: 
{'dict': ({'df':           A         B
   0  0.170801  0.895366
   1  0.838238  0.052592
   2  0.664140  0.289750
   3  0.449593  0.872087
   4  0.983618  0.744359},
  {'string': 'foo'},
  {'scalar': 1.0},
  {'s': 2013-01-01    0.548134
   2013-01-02    0.503447
   2013-01-03    0.348438
```

```
2013-01-04    0.707267
2013-01-05    0.261656
Freq: D, dtype: float64})}
```

Read/Write API

Msgpacks can also be read from and written to strings.

```
In [324]: df.to_msgpack()
Out[324]: b'\x84\xa3typ\xadblock_manager\xa5klass\xa9DataFrame\xa4axes\x92\x86
```

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

```
In [325]: pd.read_msgpack(df.to_msgpack() + s.to_msgpack())
Out[325]:
[          A         B
 0  0.170801  0.895366
 1  0.838238  0.052592
 2  0.664140  0.289750
 3  0.449593  0.872087
 4  0.983618  0.744359, 2013-01-01    0.548134
 2013-01-02    0.503447
 2013-01-03    0.348438
 2013-01-04    0.707267
 2013-01-05    0.261656
 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: As of version 0.15.0, pandas requires PyTables >= 3.0.0. Stores written with prior versions of pandas / PyTables >= 2.3 are fully compatible (this was the previous minimum PyTables required version).

Warning: There is a PyTables indexing bug 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.

Warning: As of version 0.17.0, HDFStore will not drop rows that have all missing values by default. Previously, if all values (except the index) were missing, HDFStore would not write those rows to disk.

```
In [326]: store = pd.HDFStore('store.h5')

In [327]: print(store)
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
Empty
```

Objects can be written to the file just like adding key-value pairs to a dict:

```
In [328]: np.random.seed(1234)

In [329]: index = pd.date_range('1/1/2000', periods=8)

In [330]: s = pd.Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [331]: df = pd.DataFrame(randn(8, 3), index=index,
   .....:                   columns=['A', 'B', 'C'])
   .....:

In [332]: wp = pd.Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
   .....:             major_axis=pd.date_range('1/1/2000', periods=5),
   .....:             minor_axis=['A', 'B', 'C', 'D'])
   .....:

# store.put('s', s) is an equivalent method
In [333]: store['s'] = s

In [334]: store['df'] = df

In [335]: store['wp'] = wp

# the type of stored data
In [336]: store.root.wp._v_attrs.pandas_type
Out[336]: 'wide'
```

```
In [337]: store
Out[337]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df            frame        (shape->[8,3])
/s             series       (shape->[5])
/wp            wide         (shape->[2,5,4])
```

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

```
# store.get('df') is an equivalent method
In [338]: store['df']
Out[338]:
                   A         B         C
2000-01-01  0.887163  0.859588 -0.636524
2000-01-02  0.015696 -2.242685  1.150036
2000-01-03  0.991946  0.953324 -2.021255
2000-01-04 -0.334077  0.002118  0.405453
2000-01-05  0.289092  1.321158 -1.546906
2000-01-06 -0.202646 -0.655969  0.193421
2000-01-07  0.553439  1.318152 -0.469305
2000-01-08  0.675554 -1.817027 -0.183109

# dotted (attribute) access provides get as well
In [339]: store.df
Out[339]:
                   A         B         C
2000-01-01  0.887163  0.859588 -0.636524
2000-01-02  0.015696 -2.242685  1.150036
2000-01-03  0.991946  0.953324 -2.021255
2000-01-04 -0.334077  0.002118  0.405453
2000-01-05  0.289092  1.321158 -1.546906
2000-01-06 -0.202646 -0.655969  0.193421
2000-01-07  0.553439  1.318152 -0.469305
2000-01-08  0.675554 -1.817027 -0.183109
```

Deletion of the object specified by the key

```
# store.remove('wp') is an equivalent method
In [340]: del store['wp']

In [341]: store
```

```
Out[341]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df            frame        (shape->[8,3])
/s             series       (shape->[5])
```

Closing a Store, Context Manager

```
In [342]: store.close()

In [343]: store
Out[343]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
File is CLOSED

In [344]: store.is_open
Out[344]: False

# Working with, and automatically closing the store with the context
# manager
In [345]: with pd.HDFStore('store.h5') as store:
   .....:     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. (new in 0.11.0)

```
In [346]: df_tl = pd.DataFrame(dict(A=list(range(5)), B=list(range(5))))

In [347]: df_tl.to_hdf('store_tl.h5','table',append=True)

In [348]: pd.read_hdf('store_tl.h5', 'table', where = ['index>2'])
Out[348]:
   A  B
3  3  3
4  4  4
```

As of version 0.17.0, HDFStore will no longer drop rows that are all missing by default. This behavior can be enabled by setting dropna=True.

```
In [349]: df_with_missing = pd.DataFrame({'col1':[0, np.nan, 2],
   .....:                                  'col2':[1, np.nan, np.nan]})
   .....:

In [350]: df_with_missing
Out[350]: 
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [351]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
   .....:                         format = 'table', mode='w')
   .....:

In [352]: pd.read_hdf('file.h5', 'df_with_missing')
Out[352]: 
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [353]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
   .....:                         format = 'table', mode='w', dropna=True)
   .....:

In [354]: pd.read_hdf('file.h5', 'df_with_missing')
Out[354]: 
   col1  col2
0   0.0   1.0
2   2.0   NaN
```

This is also true for the major axis of a Panel:

```
In [355]: matrix = [[[np.nan, np.nan, np.nan],[1,np.nan,np.nan]],
   .....:        [[np.nan, np.nan, np.nan], [np.nan,5,6]],
   .....:        [[np.nan, np.nan, np.nan],[np.nan,3,np.nan]]]
   .....:

In [356]: panel_with_major_axis_all_missing = pd.Panel(matrix,
   .....:         items=['Item1', 'Item2','Item3'],
```

```
   ...:        major_axis=[1,2],
   ...:        minor_axis=['A', 'B', 'C'])
   ...:

In [357]: panel_with_major_axis_all_missing
Out[357]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 2 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item3
Major_axis axis: 1 to 2
Minor_axis axis: A to C

In [358]: panel_with_major_axis_all_missing.to_hdf('file.h5', 'panel',
   ...:                              dropna = True,
   ...:                              format='table',
   ...:                              mode='w')
   ...:

In [359]: reloaded = pd.read_hdf('file.h5', 'panel')

In [360]: reloaded
Out[360]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 1 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item3
Major_axis axis: 2 to 2
Minor_axis axis: A to C
```

Fixed Format

Note: This was prior to 0.13.0 the Storer 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 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(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 & query type operations are supported. This format is specified by format='table' or format='t' to append or put or to_hdf

New in version 0.13.

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

In [362]: df1 = df[0:4]

In [363]: df2 = df[4:]

# append data (creates a table automatically)
In [364]: store.append('df', df1)

In [365]: store.append('df', df2)

In [366]: store
Out[366]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df            frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])

# select the entire object
In [367]: store.select('df')
Out[367]:
                   A         B         C
2000-01-01  0.887163  0.859588 -0.636524
2000-01-02  0.015696 -2.242685  1.150036
2000-01-03  0.991946  0.953324 -2.021255
```

```
2000-01-04 -0.334077  0.002118  0.405453
2000-01-05  0.289092  1.321158 -1.546906
2000-01-06 -0.202646 -0.655969  0.193421
2000-01-07  0.553439  1.318152 -0.469305
2000-01-08  0.675554 -1.817027 -0.183109

# the type of stored data
In [368]: store.root.df._v_attrs.pandas_type
Out[368]: '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 [369]: store.put('foo/bar/bah', df)

In [370]: store.append('food/orange', df)

In [371]: store.append('food/apple',  df)

In [372]: store
Out[372]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df                    frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah           frame        (shape->[8,3])
/food/apple            frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[inde
/food/orange           frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[ind

# a list of keys are returned
In [373]: store.keys()
Out[373]: ['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
```

```
In [374]: store.remove('food')

In [375]: store
Out[375]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df                  frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah         frame        (shape->[8,3])
```

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), 'axis1' (Ar
```

Instead, use explicit string based keys

```
In [376]: store['foo/bar/bah']
Out[376]:
                   A         B         C
2000-01-01  0.887163  0.859588 -0.636524
2000-01-02  0.015696 -2.242685  1.150036
2000-01-03  0.991946  0.953324 -2.021255
2000-01-04 -0.334077  0.002118  0.405453
2000-01-05  0.289092  1.321158 -1.546906
2000-01-06 -0.202646 -0.655969  0.193421
2000-01-07  0.553439  1.318152 -0.469305
2000-01-08  0.675554 -1.817027 -0.183109
```

Storing Types

Storing Mixed Types in a Table

Storing mixed-dtype data is supported. Strings are stored as a fixed-width using the maximum size of the appended column. Subsequent attempts at appending longer strings will raise a ValueError.

Passing min_itemsize={`values`: size} as a parameter to append will set a larger minimum for the string columns. Storing floats, strings, ints, bools, datetime64 are currently supported. For string columns, passing nan_rep = 'nan' to append will change the default nan representation on disk (which converts to/from *np.nan*), this defaults to *nan*.

```
In [377]: df_mixed = pd.DataFrame({ 'A' : randn(8),
   .....:                           'B' : randn(8),
   .....:                           'C' : np.array(randn(8),dtype='float32'),
   .....:                           'string' :'string',
   .....:                           'int' : 1,
   .....:                           'bool' : True,
   .....:                           'datetime64' : pd.Timestamp('20010102')},
   .....:                         index=list(range(8)))
   .....:

In [378]: df_mixed.loc[df_mixed.index[3:5], ['A', 'B', 'string', 'datetime64']] = np.nan

In [379]: store.append('df_mixed', df_mixed, min_itemsize = {'values': 50})

In [380]: df_mixed1 = store.select('df_mixed')

In [381]: df_mixed1
Out[381]:
          A         B         C  bool datetime64  int  string
0  0.704721 -1.152659 -0.430096  True 2001-01-02    1  string
1 -0.785435  0.631979  0.767369  True 2001-01-02    1  string
2  0.462060  0.039513  0.984920  True 2001-01-02    1  string
3       NaN       NaN  0.270836  True        NaT    1     NaN
4       NaN       NaN  1.391986  True        NaT    1     NaN
5 -0.926254  1.321106  0.079842  True 2001-01-02    1  string
6  2.007843  0.152631 -0.399965  True 2001-01-02    1  string
7  0.226963  0.164530 -1.027851  True 2001-01-02    1  string

In [382]: df_mixed1.get_dtype_counts()
Out[382]:
bool              1
datetime64[ns]    1
```

```
float32        1
float64        2
int64          1
object         1
dtype: int64

# we have provided a minimum string column size
In [383]: store.root.df_mixed.table
Out[383]:
/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}
```

Storing Multi-Index DataFrames

Storing multi-index dataframes as tables is very similar to storing/selecting from homogeneous index DataFrames.

```
In [384]: index = pd.MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
   .....:                               ['one', 'two', 'three']],
   .....:                       labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
   .....:                               [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   .....:                       names=['foo', 'bar'])
   .....:

In [385]: df_mi = pd.DataFrame(np.random.randn(10, 3), index=index,
   .....:                      columns=['A', 'B', 'C'])
   .....:

In [386]: df_mi
Out[386]:
                  A         B         C
```

```
foo bar
foo one   -0.584718  0.816594 -0.081947
    two   -0.344766  0.528288 -1.068989
    three -0.511881  0.291205  0.566534
bar one    0.503592  0.285296  0.484288
    two    1.363482 -0.781105 -0.468018
baz two    1.224574 -1.281108  0.875476
    three -1.710715 -0.450765  0.749164
qux one   -0.203933 -0.182175  0.680656
    two   -1.818499  0.047072  0.394844
    three -0.248432 -0.617707 -0.682884

In [387]: store.append('df_mi',df_mi)

In [388]: store.select('df_mi')
Out[388]: 
                  A         B         C
foo bar                                
foo one   -0.584718  0.816594 -0.081947
    two   -0.344766  0.528288 -1.068989
    three -0.511881  0.291205  0.566534
bar one    0.503592  0.285296  0.484288
    two    1.363482 -0.781105 -0.468018
baz two    1.224574 -1.281108  0.875476
    three -1.710715 -0.450765  0.749164
qux one   -0.203933 -0.182175  0.680656
    two   -1.818499  0.047072  0.394844
    three -0.248432 -0.617707 -0.682884

# the levels are automatically included as data columns
In [389]: store.select('df_mi', 'foo=bar')
Out[389]: 
                A         B         C
foo bar                              
bar one  0.503592  0.285296  0.484288
    two  1.363482 -0.781105 -0.468018
```

Querying

Querying a Table

Warning: This query capabilities have changed substantially starting in 0.13.0. Queries from prior version are accepted (with a DeprecationWarning) printed if its not string-like.

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 DataFrame
- major_axis, minor_axis, and items are supported indexers of the Panel
- 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:

- 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 [390]: dfq = pd.DataFrame(randn(10,4),columns=list('ABCD'),index=pd.date_range('

In [391]: store.append('dfq',dfq,format='table',data_columns=True)
```

Use boolean expressions, with in-line function evaluation.

```
In [392]: store.select('dfq',"index>pd.Timestamp('20130104') & columns=['A', 'B']")
Out[392]:
                   A         B
2013-01-05  1.210384  0.797435
2013-01-06 -0.850346  1.176812
2013-01-07  0.984188 -0.121728
2013-01-08  0.796595 -0.474021
2013-01-09 -0.804834 -2.123620
2013-01-10  0.334198  0.536784
```

Use and inline column reference

```
In [393]: store.select('dfq',where="A>0 or C>0")
Out[393]:
                   A         B         C         D
2013-01-01  0.436258 -1.703013  0.393711 -0.479324
2013-01-02 -0.299016  0.694103  0.678630  0.239556
2013-01-03  0.151227  0.816127  1.893534  0.639633
2013-01-04 -0.962029 -2.085266  1.930247 -1.735349
2013-01-05  1.210384  0.797435 -0.379811  0.702562
2013-01-07  0.984188 -0.121728  2.365769  0.496143
2013-01-08  0.796595 -0.474021 -0.056696  1.357797
2013-01-10  0.334198  0.536784 -0.743830 -0.320204
```

Works with a Panel as well.

```
In [394]: store.append('wp',wp)

In [395]: store
Out[395]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df                     frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/df_mi                  frame_table  (typ->appendable_multi,nrows->10,ncols->5,indexers->
/df_mixed               frame_table  (typ->appendable,nrows->8,ncols->7,indexers->[index
/dfq                    frame_table  (typ->appendable,nrows->10,ncols->4,indexers->[index],
/foo/bar/bah            frame        (shape->[8,3])
/wp                     wide_table   (typ->appendable,nrows->20,ncols->2,indexers->[major_
```

```
In [396]: store.select('wp', "major_axis>pd.Timestamp('20000102') & minor_axis=['A', '
Out[396]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to B
```

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 [397]: store.select('df', "columns=['A', 'B']")
Out[397]:
                   A         B
2000-01-01  0.887163  0.859588
2000-01-02  0.015696 -2.242685
2000-01-03  0.991946  0.953324
2000-01-04 -0.334077  0.002118
2000-01-05  0.289092  1.321158
2000-01-06 -0.202646 -0.655969
2000-01-07  0.553439  1.318152
2000-01-08  0.675554 -1.817027
```

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.

```
# this is effectively what the storage of a Panel looks like
In [398]: wp.to_frame()
Out[398]:
                     Item1     Item2
major      minor
2000-01-01 A      1.058969  0.215269
           B     -0.397840  0.841009
           C      0.337438 -1.445810
           D      1.047579 -1.401973
2000-01-02 A      1.045938 -0.100918
           B      0.863717 -0.548242
           C     -0.122092 -0.144620
...                    ...       ...
2000-01-04 B      0.036142  0.307969
           C     -2.074978 -0.208499
```

```
           D     0.247792  1.033801
2000-01-05 A    -0.897157 -2.400454
           B    -0.136795  2.030604
           C     0.018289 -1.142631
           D     0.755414  0.211883

[20 rows x 2 columns]

# limiting the search
In [399]: store.select('wp',"major_axis>20000102 & minor_axis=['A','B']",
   .....:              start=0, stop=10)
   .....:
Out[399]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 1 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-03 00:00:00
Minor_axis axis: A to B
```

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]

New in version 0.13.

Beginning in 0.13.0, 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 [400]: from datetime import timedelta

In [401]: dftd = pd.DataFrame(dict(A = pd.Timestamp('20130101'), B = [ pd.Timestamp

In [402]: dftd['C'] = dftd['A']-dftd['B']

In [403]: dftd
```

```
Out[403]: 
           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 [404]: store.append('dftd',dftd,data_columns=True)

In [405]: store.select('dftd',"C<'-3.5D'")
Out[405]: 
           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 (starting 0.10.1) 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 [406]: i = store.root.df.table.cols.index.index

In [407]: i.optlevel, i.kind
Out[407]: (6, 'medium')
```

```
# change an index by passing new parameters
In [408]: store.create_table_index('df', optlevel=9, kind='full')

In [409]: i = store.root.df.table.cols.index.index

In [410]: i.optlevel, i.kind
Out[410]: (9, 'full')
```

Oftentimes when appending large amounts of data to a store, it is useful to turn off index creation for each append, then recreate at the end.

```
In [411]: df_1 = pd.DataFrame(randn(10,2),columns=list('AB'))

In [412]: df_2 = pd.DataFrame(randn(10,2),columns=list('AB'))

In [413]: st = pd.HDFStore('appends.h5',mode='w')

In [414]: st.append('df', df_1, data_columns=['B'], index=False)

In [415]: st.append('df', df_2, data_columns=['B'], index=False)

In [416]: st.get_storer('df').table
Out[416]:
/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 [417]: st.create_table_index('df', columns=['B'], optlevel=9, kind='full')

In [418]: st.get_storer('df').table
Out[418]:
/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 [419]: 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 [420]: df_dc = df.copy()

In [421]: df_dc['string'] = 'foo'

In [422]: df_dc.loc[df_dc.index[4:6], 'string'] = np.nan

In [423]: df_dc.loc[df_dc.index[7:9], 'string'] = 'bar'

In [424]: df_dc['string2'] = 'cool'

In [425]: df_dc.loc[df_dc.index[1:3], ['B','C']] = 1.0

In [426]: df_dc
Out[426]: 
                   A         B         C string string2
2000-01-01  0.887163  0.859588 -0.636524    foo    cool
2000-01-02  0.015696  1.000000  1.000000    foo    cool
2000-01-03  0.991946  1.000000  1.000000    foo    cool
2000-01-04 -0.334077  0.002118  0.405453    foo    cool
2000-01-05  0.289092  1.321158 -1.546906    NaN    cool
2000-01-06 -0.202646 -0.655969  0.193421    NaN    cool
2000-01-07  0.553439  1.318152 -0.469305    foo    cool
2000-01-08  0.675554 -1.817027 -0.183109    bar    cool

# on-disk operations
```

```
In [427]: store.append('df_dc', df_dc, data_columns = ['B', 'C', 'string', 'string2'])

In [428]: store.select('df_dc', where='B>0')
Out[428]:
                   A         B         C string string2
2000-01-01  0.887163  0.859588 -0.636524    foo    cool
2000-01-02  0.015696  1.000000  1.000000    foo    cool
2000-01-03  0.991946  1.000000  1.000000    foo    cool
2000-01-04 -0.334077  0.002118  0.405453    foo    cool
2000-01-05  0.289092  1.321158 -1.546906    NaN    cool
2000-01-07  0.553439  1.318152 -0.469305    foo    cool

# getting creative
In [429]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')
Out[429]:
                   A         B         C string string2
2000-01-02  0.015696  1.000000  1.000000    foo    cool
2000-01-03  0.991946  1.000000  1.000000    foo    cool
2000-01-04 -0.334077  0.002118  0.405453    foo    cool

# this is in-memory version of this type of selection
In [430]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]
Out[430]:
                   A         B         C string string2
2000-01-02  0.015696  1.000000  1.000000    foo    cool
2000-01-03  0.991946  1.000000  1.000000    foo    cool
2000-01-04 -0.334077  0.002118  0.405453    foo    cool

# we have automagically created this index and the B/C/string/string2
# columns are stored separately as ``PyTables`` columns
In [431]: store.root.df_dc.table
Out[431]:
/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}
```

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

Starting in 0.11.0, 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 [432]: for df in store.select('df', chunksize=3):
   .....:     print(df)
   .....:
                   A         B         C
2000-01-01  0.887163  0.859588 -0.636524
2000-01-02  0.015696 -2.242685  1.150036
2000-01-03  0.991946  0.953324 -2.021255
                   A         B         C
2000-01-04 -0.334077  0.002118  0.405453
2000-01-05  0.289092  1.321158 -1.546906
2000-01-06 -0.202646 -0.655969  0.193421
                   A         B         C
2000-01-07  0.553439  1.318152 -0.469305
2000-01-08  0.675554 -1.817027 -0.183109
```

Note:

New in version 0.12.0.

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

In [434]: dfeq
Out[434]: 
   number
0       1
1       2
2       3
3       4
4       5
5       6
6       7
7       8
8       9
9      10

In [435]: store.append('dfeq', dfeq, data_columns=['number'])

In [436]: def chunks(l, n):
   .....:     return [l[i:i+n] for i in range(0, len(l), n)]
   .....: 

In [437]: evens = [2,4,6,8,10]

In [438]: coordinates = store.select_as_coordinates('dfeq','number=evens')
```

```
In [439]: for c in chunks(coordinates, 2):
   .....:     print store.select('dfeq',where=c)
   .....:
  File "<ipython-input-439-8285a1f175a6>", line 2
    print store.select('dfeq',where=c)
              ^
SyntaxError: invalid syntax
```

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 [440]: store.select_column('df_dc', 'index')
Out[440]:
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 [441]: store.select_column('df_dc', 'string')
Out[441]:
0    foo
1    foo
2    foo
3    foo
4    NaN
5    NaN
6    foo
7    bar
Name: string, dtype: object
```

Selecting coordinates

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

```
In [442]: df_coord = pd.DataFrame(np.random.randn(1000,2),index=pd.date_range('2(

In [443]: store.append('df_coord',df_coord)

In [444]: c = store.select_as_coordinates('df_coord','index>20020101')

In [445]: c.summary()
Out[445]: 'Int64Index: 268 entries, 732 to 999'

In [446]: store.select('df_coord',where=c)
Out[446]:
                   0         1
2002-01-02 -0.178266 -0.064638
2002-01-03 -1.204956 -3.880898
2002-01-04  0.974470  0.415160
2002-01-05  1.751967  0.485011
2002-01-06 -0.170894  0.748870
2002-01-07  0.629793  0.811053
2002-01-08  2.133776  0.238459
...              ...       ...
2002-09-20 -0.181434  0.612399
2002-09-21 -0.763324 -0.354962
2002-09-22 -0.261776  0.812126
2002-09-23  0.482615 -0.886512
2002-09-24 -0.037757 -0.562953
2002-09-25  0.897706  0.383232
2002-09-26 -1.324806  1.139269

[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 [447]: df_mask = pd.DataFrame(np.random.randn(1000,2),index=pd.date_range('20

In [448]: store.append('df_mask',df_mask)

In [449]: c = store.select_column('df_mask','index')

In [450]: where = c[pd.DatetimeIndex(c).month==5].index

In [451]: store.select('df_mask',where=where)
Out[451]:
                   0         1
2000-05-01 -1.006245 -0.616759
2000-05-02  0.218940  0.717838
2000-05-03  0.013333  1.348060
2000-05-04  0.662176 -1.050645
2000-05-05 -1.034870 -0.243242
2000-05-06 -0.753366 -1.454329
2000-05-07 -1.022920 -0.476989
...              ...       ...
2002-05-25 -0.509090 -0.389376
2002-05-26  0.150674  1.164337
2002-05-27 -0.332944  0.115181
2002-05-28 -1.048127 -0.605733
2002-05-29  1.418754 -0.442835
2002-05-30 -0.433200  0.835001
2002-05-31 -1.041278  1.401811

[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 [452]: store.get_storer('df_dc').nrows
Out[452]: 8
```

Multiple Table Queries

New in 0.10.1 are the methods append_to_multiple and select_as_multiple, that 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 [453]: df_mt = pd.DataFrame(randn(8, 6), index=pd.date_range('1/1/2000', periods=
   .....:                    columns=['A', 'B', 'C', 'D', 'E', 'F'])
   .....:

In [454]: df_mt['foo'] = 'bar'

In [455]: df_mt.loc[df_mt.index[1], ('A', 'B')] = np.nan

# you can also create the tables individually
In [456]: store.append_to_multiple({'df1_mt': ['A', 'B'], 'df2_mt': None },
   .....:                          df_mt, selector='df1_mt')
   .....:

In [457]: store
Out[457]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df              frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])
/df1_mt          frame_table  (typ->appendable,nrows->8,ncols->2,indexers->[index]
/df2_mt          frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index]
/df_coord        frame_table  (typ->appendable,nrows->1000,ncols->2,indexers->[in
/df_dc           frame_table  (typ->appendable,nrows->8,ncols->5,indexers->[index],
/df_mask         frame_table  (typ->appendable,nrows->1000,ncols->2,indexers->[in
/df_mi           frame_table  (typ->appendable_multi,nrows->10,ncols->5,indexers->
/df_mixed        frame_table  (typ->appendable,nrows->8,ncols->7,indexers->[index
```

```
/dfeq             frame_table  (typ->appendable,nrows->10,ncols->1,indexers->[index]
/dfq              frame_table  (typ->appendable,nrows->10,ncols->4,indexers->[index],
/dftd             frame_table  (typ->appendable,nrows->10,ncols->3,indexers->[index],
/foo/bar/bah          frame       (shape->[8,3])
/wp               wide_table   (typ->appendable,nrows->20,ncols->2,indexers->[major_

# individual tables were created
In [458]: store.select('df1_mt')
Out[458]:
                   A         B
2000-01-01  0.714697  0.318215
2000-01-02       NaN       NaN
2000-01-03 -0.086919  0.416905
2000-01-04  0.489131 -0.253340
2000-01-05 -0.382952 -0.397373
2000-01-06  0.538116  0.226388
2000-01-07 -2.073479 -0.115926
2000-01-08 -0.695400  0.402493

In [459]: store.select('df2_mt')
Out[459]:
                   C         D         E         F  foo
2000-01-01  0.607460  0.790907  0.852225  0.096696  bar
2000-01-02  0.811031 -0.356817  1.047085  0.664705  bar
2000-01-03 -0.764381 -0.287229 -0.089351 -1.035115  bar
2000-01-04 -1.948100 -0.116556  0.800597 -0.796154  bar
2000-01-05 -0.717627  0.156995 -0.344718 -0.171208  bar
2000-01-06  1.541729  0.205256  1.998065  0.953591  bar
2000-01-07  1.391070  0.303013  1.093347 -0.101000  bar
2000-01-08 -1.507639  0.089575  0.658822 -1.037627  bar

# as a multiple
In [460]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'],
   .....:                           selector = 'df1_mt')
   .....:
Out[460]:
                   A         B         C         D         E         F  foo
2000-01-01  0.714697  0.318215  0.607460  0.790907  0.852225  0.096696  bar
2000-01-06  0.538116  0.226388  1.541729  0.205256  1.998065  0.953591  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. This is especially true in higher dimensional objects (Panel and Panel4D). To get optimal performance, it's worthwhile to have the dimension you are deleting be the first of the indexables.

Data is ordered (on the disk) in terms of the indexables. Here's a simple use case. You store panel-type data, with dates in the major_axis and ids in the minor_axis. The data is then interleaved like this:

- date_1 - id_1 - id_2 - . - id_n
- date_2 - id_1 - . - id_n

It should be clear that a delete operation on the major_axis will be fairly quick, as one chunk is 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.

```
# returns the number of rows deleted
In [461]: store.remove('wp', 'major_axis>20000102' )
Out[461]: 12

In [462]: store.select('wp')
Out[462]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-02 00:00:00
Minor_axis axis: A to D
```

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.

- Pass complevel=int for a compression level (1-9, with 0 being no compression, and the default)
- Pass complib=lib where lib is any of zlib, bzip2, lzo, blosc for whichever compression library you prefer.

HDFStore will use the file based compression scheme if no overriding complib or complevel options are provided. blosc offers very fast compression, and is my most used. Note that lzo and bzip2 may not be installed (by Python) by default.

Compression for all objects within the file

```
store_compressed = pd.HDFStore('store_compressed.h5', complevel=9, complib='blos
```

Or on-the-fly compression (this only applies to tables). You can turn off file compression for a specific table by passing complevel=0

```
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.l
```

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 its items (Panel) / columns (DataFrame) are fixed; only exactly the same columns can be appended
- Be aware that timezones (e.g., pytz.timezone('US/Eastern')) are not necessarily equal across timezone versions. So if data is localized to a specific timezone in the HDFStore using one version of a timezone library and that data is updated with another version, the data will be converted to UTC since these timezones are not considered equal. Either use the same version of timezone library or use tz_convert with the updated timezone definition.

Warning: PyTables will show a NaturalNameWarning if a column name cannot be used as an attribute selector. *Natural* identifiers contain only letters, numbers, and underscores, and may not begin with a number. Other identifiers cannot be used in a where clause and are generally a bad idea.

DataTypes

HDFStore will map an object dtype to the PyTables underlying dtype. This means the following types are known to work:

Type	Represents missing values
floating : float64, float32, float16	np.nan
integer : int64, int32, int8, uint64,uint32, uint8	
boolean	
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

New in version 0.15.2.

Writing data to a HDFStore that contains a category dtype was implemented in 0.15.2. Queries work the same as if it was an object array. However, the category dtyped data is stored in a more efficient manner.

```
In [463]: dfcat = pd.DataFrame({ 'A' : pd.Series(list('aabbcdba')).astype('category'),
   .....:                       'B' : np.random.randn(8) })
   .....:

In [464]: dfcat
Out[464]:
   A         B
0  a  0.603273
1  a  0.262554
2  b -0.979586
3  b  2.132387
4  c  0.892485
5  d  1.996474
6  b  0.231425
7  a  0.980070

In [465]: dfcat.dtypes
Out[465]:
A    category
B     float64
dtype: object

In [466]: cstore = pd.HDFStore('cats.h5', mode='w')

In [467]: cstore.append('dfcat', dfcat, format='table', data_columns=['A'])

In [468]: result = cstore.select('dfcat', where="A in ['b','c']")

In [469]: result
Out[469]:
   A         B
2  b -0.979586
3  b  2.132387
4  c  0.892485
6  b  0.231425

In [470]: result.dtypes
Out[470]:
```

```
A    category
B     float64
dtype: object
```

Warning: The format of the Categorical is readable by prior versions of pandas (< 0.15.2), but will retrieve the data as an integer based column (e.g. the codes). However, the categories *can* be retrieved but require the user to select them manually using the explicit meta path.

The data is stored like so:

```
In [471]: cstore
Out[471]:
<class 'pandas.io.pytables.HDFStore'>
File path: cats.h5
/dfcat                    frame_table  (typ->appendable,nrows->8,ncols->2,indexers->[in
/dfcat/meta/A/meta           series_table (typ->appendable,nrows->4,ncols->1,indexe

# to get the categories
In [472]: cstore.select('dfcat/meta/A/meta')
Out[472]:
0    a
1    b
2    c
3    d
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.

Starting in 0.11.0, 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 [473]: dfs = pd.DataFrame(dict(A = 'foo', B = 'bar'),index=list(range(5)))

In [474]: dfs
Out[474]:
     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 [475]: store.append('dfs', dfs, min_itemsize = 30)

In [476]: store.get_storer('dfs').table
Out[476]:
/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 [477]: store.append('dfs2', dfs, min_itemsize = { 'A' : 30 })

In [478]: store.get_storer('dfs2').table
Out[478]:
/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 [479]: dfss = pd.DataFrame(dict(A = ['foo','bar','nan']))

In [480]: dfss
Out[480]:
     A
0  foo
1  bar
2  nan

In [481]: store.append('dfss', dfss)

In [482]: store.select('dfss')
Out[482]:
     A
0  foo
1  bar
2  NaN

# here you need to specify a different nan rep
In [483]: store.append('dfss2', dfss, nan_rep='_nan_')

In [484]: store.select('dfss2')
Out[484]:
     A
0  foo
1  bar
2  nan
```

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 [485]: np.random.seed(1)

In [486]: 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 [487]: df_for_r.head()
Out[487]: 
   class     first    second
0      0  0.417022  0.326645
1      0  0.720324  0.527058
2      1  0.000114  0.885942
3      1  0.302333  0.357270
4      1  0.146756  0.908535

In [488]: store_export = pd.HDFStore('export.h5')

In [489]: store_export.append('df_for_r', df_for_r, data_columns=df_dc.columns)

In [490]: store_export
Out[490]: 
<class 'pandas.io.pytables.HDFStore'>
File path: export.h5
/df_for_r            frame_table  (typ->appendable,nrows->100,ncols->3,indexers->[index]
```

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
  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.

Backwards Compatibility

0.10.1 of HDFStore can read tables created in a prior version of pandas, however query terms using the prior (undocumented) methodology are unsupported. HDFStore will issue a warning if you try to use a legacy-format file. You must read in the entire file and write it out using the new format, using the method copy to take advantage of the updates. The group attribute pandas_version contains the version information. copy takes a number of options, please see the docstring.

```
# a legacy store
In [491]: legacy_store = pd.HDFStore(legacy_file_path,'r')

In [492]: legacy_store
Out[492]:
<class 'pandas.io.pytables.HDFStore'>
File path: /Users/taugspurger/Envs/pandas-dev/lib/python3.6/site-packages/pandas/
/a                series       (shape->[30])
/b                frame        (shape->[30,4])
/df1_mixed            frame_table [0.10.0] (typ->appendable,nrows->30,ncols->11,indexe
/foo/bar           wide        (shape->[3,30,4])
/p1_mixed            wide_table  [0.10.0] (typ->appendable,nrows->120,ncols->9,indexer
/p4d_mixed            ndim_table  [0.10.0] (typ->appendable,nrows->360,ncols->9,indexe

# copy (and return the new handle)
In [493]: new_store = legacy_store.copy('store_new.h5')

In [494]: new_store
Out[494]:
<class 'pandas.io.pytables.HDFStore'>
File path: store_new.h5
/a                series       (shape->[30])
/b                frame        (shape->[30,4])
/df1_mixed            frame_table  (typ->appendable,nrows->30,ncols->11,indexers->[ind
/foo/bar           wide        (shape->[3,30,4])
/p1_mixed            wide_table   (typ->appendable,nrows->120,ncols->9,indexers->[maj
/p4d_mixed            wide_table   (typ->appendable,nrows->360,ncols->9,indexers->[iter

In [495]: new_store.close()
```

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.

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 simply .reset_index() in order to store the index.
- 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 [496]: 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 [497]: df
Out[497]:
   a  b  c    d      e  f          g                         h          i
0  a  1  3  4.0   True  a 2013-01-01 2013-01-01 00:00:00-05:00 2013-01-01
1  b  2  4  5.0  False  b 2013-01-02 2013-01-02 00:00:00-05:00 2013-01-01
2  c  3  5  6.0   True  c 2013-01-03 2013-01-03 00:00:00-05:00 2013-01-01

In [498]: df.dtypes
Out[498]:
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 [499]: df.to_feather('example.feather')
```

Read from a feather file.

```
In [500]: result = pd.read_feather('example.feather')

In [501]: result
Out[501]:
```

```
   a  b  c    d      e  f          g                          h          i
0  a  1  3  4.0   True  a 2013-01-01 2013-01-01 00:00:00-05:00 2013-01-01
1  b  2  4  5.0  False  b 2013-01-02 2013-01-02 00:00:00-05:00 2013-01-01
2  c  3  5  6.0   True  c 2013-01-03 2013-01-03 00:00:00-05:00 2013-01-01

# we preserve dtypes
In [502]: result.dtypes
Out[502]: 
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
```

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.

New in version 0.14.0.

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(name, con[, flavor, ...])	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

```
In [503]: from sqlalchemy import create_engine

# Create your engine.
In [504]: 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 [505]: 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 [506]: 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:

```
In [507]: from sqlalchemy.types import String

In [508]: 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.

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 [509]: pd.read_sql_table('data', engine)
Out[509]: 
   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 [510]: pd.read_sql_table('data', engine, index_col='id')
Out[510]: 
    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 [511]: pd.read_sql_table('data', engine, columns=['Col_1', 'Col_2'])
Out[511]: 
  Col_1  Col_2
0     X  27.50
1     Y -12.50
2     Z   5.73
```

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

```
In [512]: pd.read_sql_table('data', engine, parse_dates=['Date'])
Out[512]: 
   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:%
```

You can check if a table exists using has_table()

Schema support

New in version 0.15.0.

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 [513]: pd.read_sql_query('SELECT * FROM data', engine)
Out[513]: 
   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 [514]: pd.read_sql_query("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;", engi
Out[514]: 
   id Col_1  Col_2
0  42     Y  -12.5
```

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

```
In [515]: df = pd.DataFrame(np.random.randn(20, 3), columns=list('abc'))
```

```
In [516]: df.to_sql('data_chunks', engine, index=False)
```

```
In [517]: for chunk in pd.read_sql_query("SELECT * FROM data_chunks", engine, chun
   .....:     print(chunk)
   .....:
          a         b         c
0  0.280665 -0.073113  1.160339
1  0.369493  1.904659  1.111057
2  0.659050 -1.627438  0.602319
3  0.420282  0.810952  1.044442
4 -0.400878  0.824006 -0.562305
          a         b         c
0  1.954878 -1.331952 -1.760689
1 -1.650721 -0.890556 -1.119115
2  1.956079 -0.326499 -1.342676
3  1.114383 -0.586524 -1.236853
4  0.875839  0.623362 -0.434957
          a         b         c
0  1.407540  0.129102  1.616950
1  0.502741  1.558806  0.109403
2 -1.219744  2.449369 -0.545774
3 -0.198838 -0.700399 -0.203394
4  0.242669  0.201830  0.661020
          a         b         c
0  1.792158 -0.120465 -1.233121
1 -1.182318 -0.665755 -1.674196
2  0.825030 -0.498214 -0.310985
3 -0.001891 -1.396620 -0.861316
4  0.674712  0.618539 -0.443172
```

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, T
```

Engine connection examples

To connect with SQLAlchemy you use the create_engine() function to create an engine object from database URI. You only need to create the engine once per database you are connecting to.

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')

engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')

engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

engine = create_engine('mssql+pyodbc://mydsn')

# sqlite://<nohostname>/<path>
# where <path> is relative:
engine = create_engine('sqlite:///foo.db')

# or absolute, starting with a slash:
engine = create_engine('sqlite:////absolute/path/to/foo.db')
```

For more information see the examples the SQLAlchemy documentation

Advanced SQLAlchemy queries

You can use SQLAlchemy constructs to describe your query.

Use sqlalchemy.text() to specify query parameters in a backend-neutral way

```
In [518]: import sqlalchemy as sa

In [519]: pd.read_sql(sa.text('SELECT * FROM data where Col_1=:col1'), engine, param
Out[519]:
   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 [520]: metadata = sa.MetaData()
```

```
In [521]: 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 [522]: pd.read_sql(sa.select([data_table]).where(data_table.c.Col_3 == True), engine
Out[522]: 
   index       Date Col_1  Col_2  Col_3
0      0 2010-10-18     X  27.50   True
1      2 2010-10-20     Z   5.73   True
```

You can combine SQLAlchemy expressions with parameters passed to read_sql() using sqlalchemy.bindparam()

```
In [523]: import datetime as dt

In [524]: expr = sa.select([data_table]).where(data_table.c.Date > sa.bindparam('date')

In [525]: pd.read_sql(expr, engine, params={'date': dt.datetime(2010, 10, 18)})
Out[525]: 
   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', cnx)
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 cocumentation can be found here

Stata Format

New in version 0.12.0.

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).

```
In [526]: df = pd.DataFrame(randn(10, 2), columns=list('AB'))

In [527]: 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 [528]: pd.read_stata('stata.dta')
Out[528]:
   index         A         B
0      0  1.810535 -1.305727
1      1 -0.344987 -0.230840
2      2 -2.793085  1.937529
3      3  0.366332 -1.044589
4      4  2.051173  0.585662
5      5  0.429526 -0.606998
6      6  0.106223 -1.525680
7      7  0.795026 -0.374438
8      8  0.134048  1.202055
9      9  0.284748  0.262467
```

New in version 0.16.0.

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.

```
In [529]: reader = pd.read_stata('stata.dta', chunksize=3)
```

```
In [530]: 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 [531]: reader = pd.read_stata('stata.dta', iterator=True)

In [532]: chunk1 = reader.read(5)

In [533]: 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

New in version 0.15.2.

Categorical data can be exported to *Stata* data files as value labeled data. The exported data consists of the underlying category codes as integer data values and the categories as value labels. *Stata* does not have an explicit equivalent to a Categorical and information about *whether* the variable is ordered is lost when exporting.

Warning: *Stata* only supports string value labels, and so str is called on the categories when exporting data. Exporting Categorical variables with non-string categories produces a warning, and can result a loss of information if the str representations of the categories are not unique.

Labeled data can similarly be imported from *Stata* data files as Categorical variables using the keyword argument convert_categoricals (True by default). The keyword argument order_categoricals (True by default) determines whether imported Categorical variables are ordered.

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

New in version 0.17.0.

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

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:

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

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 0.13.1.

```
In [1]: df = pd.DataFrame(randn(1000000,2),columns=list('AB'))
```

```
In [2]: df.info()
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000000 entries, 0 to 999999
Data columns (total 2 columns):
A    1000000 non-null float64
B    1000000 non-null float64
dtypes: float64(2)
memory usage: 22.9 MB
```

Writing

```
In [14]: %timeit test_sql_write(df)
1 loops, best of 3: 6.24 s per loop

In [15]: %timeit test_hdf_fixed_write(df)
1 loops, best of 3: 237 ms per loop

In [26]: %timeit test_hdf_fixed_write_compress(df)
1 loops, best of 3: 245 ms per loop

In [16]: %timeit test_hdf_table_write(df)
1 loops, best of 3: 901 ms per loop

In [27]: %timeit test_hdf_table_write_compress(df)
1 loops, best of 3: 952 ms per loop

In [17]: %timeit test_csv_write(df)
1 loops, best of 3: 3.44 s per loop
```

Reading

```
In [18]: %timeit test_sql_read()
1 loops, best of 3: 766 ms per loop

In [19]: %timeit test_hdf_fixed_read()
10 loops, best of 3: 19.1 ms per loop

In [28]: %timeit test_hdf_fixed_read_compress()
10 loops, best of 3: 36.3 ms per loop

In [20]: %timeit test_hdf_table_read()
```

```
10 loops, best of 3: 39 ms per loop

In [29]: %timeit test_hdf_table_read_compress()
10 loops, best of 3: 60.6 ms per loop

In [22]: %timeit test_csv_read()
1 loops, best of 3: 620 ms per loop
```

Space on disk (in bytes)

```
25843712 Apr  8 14:11 test.sql
24007368 Apr  8 14:11 test_fixed.hdf
15580682 Apr  8 14:11 test_fixed_compress.hdf
24458444 Apr  8 14:11 test_table.hdf
16797283 Apr  8 14:11 test_table_compress.hdf
46152810 Apr  8 14:11 test.csv
```

And here's the code

```
import sqlite3
import os
from pandas.io import sql

df = pd.DataFrame(randn(1000000,2),columns=list('AB'))

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()

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)
```