```
In [324]: df.columns = pd.MultiIndex.from_product([['a'], ['b', 'd']],
   .....:                                         names=['c1', 'c2'])
   .....:

In [325]: df.to_excel('path_to_file.xlsx')

In [326]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1], header=[0, 1])

In [327]: df
Out[327]:
c1          a
c2          b  d
lvl1 lvl2
a    c      1  5
     d      2  6
b    c      3  7
     d      4  8
```

Parsing specific columns

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

Deprecated since version 0.24.0.

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

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

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

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

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

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

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

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

New in version 0.24.

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

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

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

New in version 0.24.

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

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

Parsing dates

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

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

Cell converters

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

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

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

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


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

Dtype specifications

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

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

Writing Excel files

Writing Excel files to disk

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

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

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

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

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

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

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

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

Writing Excel files to memory

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

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

bio = BytesIO()

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

# Save the workbook
writer.save()

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

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

Excel writer engines

Pandas chooses an Excel writer via two methods:

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

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

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

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

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

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

# Or via pandas configuration.
from pandas import options  # noqa: E402
options.io.excel.xlsx.writer = 'xlsxwriter'

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

Style and formatting

The look and feel of Excel worksheets created from pandas can be modified using the following parameters on the `DataFrame`'s `to_excel` method.

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

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

2.1.5 OpenDocument Spreadsheets

New in version 0.25.

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

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

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

2.1.6 Binary Excel (.xlsb) files

New in version 1.0.0.

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

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

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

2.1.7 Clipboard

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

2.1.8 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 [328]: df
Out[328]:
c1          a
c2          b  d
lvl1 lvl2
a    c      1  5
     d      2  6
b    c      3  7
     d      4  8

In [329]: 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 [330]: pd.read_pickle('foo.pkl')
Out[330]:
c1          a
c2          b  d
lvl1 lvl2
a    c      1  5
     d      2  6
b    c      3  7
     d      4  8
```

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

See: https://docs.python.org/3/library/pickle.html

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

Compressed pickle files

read_pickle(), *DataFrame.to_pickle()* and *Series.to_pickle()* can read and write compressed pickle files. The compression types of gzip, bz2, xz are supported for reading and writing. The zip file format only supports reading and must contain only one data file to be read.

The compression type can be an explicit parameter or be inferred from the file extension. If 'infer', then use gzip, bz2, zip, or xz if filename ends in '.gz', '.bz2', '.zip', or '.xz', respectively.

```
In [331]: df = pd.DataFrame({
   .....:     'A': np.random.randn(1000),
   .....:     'B': 'foo',
   .....:     'C': pd.date_range('20130101', periods=1000, freq='s')})
   .....:

In [332]: df
Out[332]:
            A    B                   C
0   -0.288267  foo 2013-01-01 00:00:00
1   -0.084905  foo 2013-01-01 00:00:01
2    0.004772  foo 2013-01-01 00:00:02
3    1.382989  foo 2013-01-01 00:00:03
4    0.343635  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.220893  foo 2013-01-01 00:16:35
996  0.492996  foo 2013-01-01 00:16:36
997 -0.461625  foo 2013-01-01 00:16:37
998  1.361779  foo 2013-01-01 00:16:38
999 -1.197988  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Using an explicit compression type:

```
In [333]: df.to_pickle("data.pkl.compress", compression="gzip")

In [334]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [335]: rt
Out[335]:
            A    B                   C
0   -0.288267  foo 2013-01-01 00:00:00
1   -0.084905  foo 2013-01-01 00:00:01
2    0.004772  foo 2013-01-01 00:00:02
3    1.382989  foo 2013-01-01 00:00:03
4    0.343635  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.220893  foo 2013-01-01 00:16:35
996  0.492996  foo 2013-01-01 00:16:36
997 -0.461625  foo 2013-01-01 00:16:37
998  1.361779  foo 2013-01-01 00:16:38
999 -1.197988  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

Inferring compression type from the extension:

```
In [336]: df.to_pickle("data.pkl.xz", compression="infer")

In [337]: rt = pd.read_pickle("data.pkl.xz", compression="infer")

In [338]: rt
Out[338]:
            A    B                   C
0   -0.288267  foo 2013-01-01 00:00:00
1   -0.084905  foo 2013-01-01 00:00:01
2    0.004772  foo 2013-01-01 00:00:02
3    1.382989  foo 2013-01-01 00:00:03
4    0.343635  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.220893  foo 2013-01-01 00:16:35
996  0.492996  foo 2013-01-01 00:16:36
997 -0.461625  foo 2013-01-01 00:16:37
998  1.361779  foo 2013-01-01 00:16:38
999 -1.197988  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]
```

The default is to 'infer':

```
In [339]: df.to_pickle("data.pkl.gz")

In [340]: rt = pd.read_pickle("data.pkl.gz")

In [341]: rt
Out[341]:
            A    B                   C
0   -0.288267  foo 2013-01-01 00:00:00
1   -0.084905  foo 2013-01-01 00:00:01
2    0.004772  foo 2013-01-01 00:00:02
3    1.382989  foo 2013-01-01 00:00:03
4    0.343635  foo 2013-01-01 00:00:04
..        ...  ...                 ...
995 -0.220893  foo 2013-01-01 00:16:35
996  0.492996  foo 2013-01-01 00:16:36
997 -0.461625  foo 2013-01-01 00:16:37
998  1.361779  foo 2013-01-01 00:16:38
999 -1.197988  foo 2013-01-01 00:16:39

[1000 rows x 3 columns]

In [342]: df["A"].to_pickle("s1.pkl.bz2")

In [343]: rt = pd.read_pickle("s1.pkl.bz2")

In [344]: rt
Out[344]:
0     -0.288267
1     -0.084905
2      0.004772
3      1.382989
4      0.343635
         ...
995   -0.220893
```

(continues on next page)

(continued from previous page)

```
996    0.492996
997   -0.461625
998    1.361779
999   -1.197988
Name: A, Length: 1000, dtype: float64
```

2.1.9 msgpack

pandas support for `msgpack` has been removed in version 1.0.0. It is recommended to use pyarrow for on-the-wire transmission of pandas objects.

Example pyarrow usage:

```
>>> import pandas as pd
>>> import pyarrow as pa
>>> df = pd.DataFrame({'A': [1, 2, 3]})
>>> context = pa.default_serialization_context()
>>> df_bytestring = context.serialize(df).to_buffer().to_pybytes()
```

For documentation on pyarrow, see here.

2.1.10 HDF5 (PyTables)

`HDFStore` is a dict-like object which reads and writes pandas using the high performance HDF5 format using the excellent PyTables library. See the *cookbook* for some advanced strategies

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

```
In [345]: store = pd.HDFStore('store.h5')

In [346]: print(store)
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

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

```
In [347]: index = pd.date_range('1/1/2000', periods=8)

In [348]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [349]: df = pd.DataFrame(np.random.randn(8, 3), index=index,
   .....:                   columns=['A', 'B', 'C'])
   .....:

# store.put('s', s) is an equivalent method
In [350]: store['s'] = s

In [351]: store['df'] = df
```

(continues on next page)

(continued from previous page)

```
In [352]: store
Out[352]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

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

```
# store.get('df') is an equivalent method
In [353]: store['df']
Out[353]:
                   A         B         C
2000-01-01  1.334065  0.521036  0.930384
2000-01-02 -1.613932  1.088104 -0.632963
2000-01-03 -0.585314 -0.275038 -0.937512
2000-01-04  0.632369 -1.249657  0.975593
2000-01-05  1.060617 -0.143682  0.218423
2000-01-06  3.050329  1.317933 -0.963725
2000-01-07 -0.539452 -0.771133  0.023751
2000-01-08  0.649464 -1.736427  0.197288

# dotted (attribute) access provides get as well
In [354]: store.df
Out[354]:
                   A         B         C
2000-01-01  1.334065  0.521036  0.930384
2000-01-02 -1.613932  1.088104 -0.632963
2000-01-03 -0.585314 -0.275038 -0.937512
2000-01-04  0.632369 -1.249657  0.975593
2000-01-05  1.060617 -0.143682  0.218423
2000-01-06  3.050329  1.317933 -0.963725
2000-01-07 -0.539452 -0.771133  0.023751
2000-01-08  0.649464 -1.736427  0.197288
```

Deletion of the object specified by the key:

```
# store.remove('df') is an equivalent method
In [355]: del store['df']

In [356]: store
Out[356]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Closing a Store and using a context manager:

```
In [357]: store.close()

In [358]: store
Out[358]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

In [359]: store.is_open
Out[359]: False

# Working with, and automatically closing the store using a context manager
In [360]: with pd.HDFStore('store.h5') as store:
```

(continues on next page)

(continued from previous page)

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

Read/write API

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

```
In [361]: df_tl = pd.DataFrame({'A': list(range(5)), 'B': list(range(5))})

In [362]: df_tl.to_hdf('store_tl.h5', 'table', append=True)

In [363]: pd.read_hdf('store_tl.h5', 'table', where=['index>2'])
Out[363]:
   A  B
3  3  3
4  4  4
```

HDFStore will by default not drop rows that are all missing. This behavior can be changed by setting `dropna=True`.

```
In [364]: df_with_missing = pd.DataFrame({'col1': [0, np.nan, 2],
   .....:                                 'col2': [1, np.nan, np.nan]})
   .....:

In [365]: df_with_missing
Out[365]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [366]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
   .....:                        format='table', mode='w')
   .....:

In [367]: pd.read_hdf('file.h5', 'df_with_missing')
Out[367]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN

In [368]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
   .....:                        format='table', mode='w', dropna=True)
   .....:

In [369]: pd.read_hdf('file.h5', 'df_with_missing')
Out[369]:
   col1  col2
0   0.0   1.0
1   NaN   NaN
2   2.0   NaN
```

Fixed format

The examples above show storing using `put`, which write the HDF5 to `PyTables` in a fixed array format, called the `fixed` format. These types of stores are **not** appendable once written (though you can simply remove them and rewrite). Nor are they **queryable**; they must be retrieved in their entirety. They also do not support dataframes with non-unique column names. The `fixed` format stores offer very fast writing and slightly faster reading than `table` stores. This format is specified by default when using `put` or `to_hdf` or by `format='fixed'` or `format='f'`.

Warning: A `fixed` format will raise a `TypeError` if you try to retrieve using a `where`:

```
>>> pd.DataFrame(np.random.randn(10, 2)).to_hdf('test_fixed.h5', 'df')
>>> pd.read_hdf('test_fixed.h5', 'df', where='index>5')
TypeError: cannot pass a where specification when reading a fixed format.
           this store must be selected in its entirety
```

Table format

`HDFStore` supports another `PyTables` format on disk, the `table` format. Conceptually a `table` is shaped very much like a DataFrame, with rows and columns. A `table` may be appended to in the same or other sessions. In addition, delete and query type operations are supported. This format is specified by `format='table'` or `format='t'` to `append` or `put` or `to_hdf`.

This format can be set as an option as well `pd.set_option('io.hdf.default_format','table')` to enable `put/append/to_hdf` to by default store in the `table` format.

```
In [370]: store = pd.HDFStore('store.h5')

In [371]: df1 = df[0:4]

In [372]: df2 = df[4:]

# append data (creates a table automatically)
In [373]: store.append('df', df1)

In [374]: store.append('df', df2)

In [375]: store
Out[375]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# select the entire object
In [376]: store.select('df')
Out[376]:
                   A         B         C
2000-01-01  1.334065  0.521036  0.930384
2000-01-02 -1.613932  1.088104 -0.632963
2000-01-03 -0.585314 -0.275038 -0.937512
2000-01-04  0.632369 -1.249657  0.975593
2000-01-05  1.060617 -0.143682  0.218423
2000-01-06  3.050329  1.317933 -0.963725
2000-01-07 -0.539452 -0.771133  0.023751
2000-01-08  0.649464 -1.736427  0.197288

# the type of stored data
```

(continues on next page)

(continued from previous page)

```
In [377]: store.root.df._v_attrs.pandas_type
Out[377]: 'frame_table'
```

Note: You can also create a `table` by passing `format='table'` or `format='t'` to a `put` operation.

Hierarchical keys

Keys to a store can be specified as a string. These can be in a hierarchical path-name like format (e.g. `foo/bar/bah`), which will generate a hierarchy of sub-stores (or `Groups` in PyTables parlance). Keys can be specified without the leading '/' and are **always** absolute (e.g. 'foo' refers to '/foo'). Removal operations can remove everything in the sub-store and **below**, so be *careful*.

```
In [378]: store.put('foo/bar/bah', df)

In [379]: store.append('food/orange', df)

In [380]: store.append('food/apple', df)

In [381]: store
Out[381]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# a list of keys are returned
In [382]: store.keys()
Out[382]: ['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [383]: store.remove('food')

In [384]: store
Out[384]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

You can walk through the group hierarchy using the `walk` method which will yield a tuple for each group key along with the relative keys of its contents.

New in version 0.24.0.

```
In [385]: for (path, subgroups, subkeys) in store.walk():
   .....:     for subgroup in subgroups:
   .....:         print('GROUP: {}/{}'.format(path, subgroup))
   .....:     for subkey in subkeys:
   .....:         key = '/'.join([path, subkey])
   .....:         print('KEY: {}'.format(key))
   .....:         print(store.get(key))
   .....:
GROUP: /foo
KEY: /df
                   A         B         C
2000-01-01  1.334065  0.521036  0.930384
2000-01-02 -1.613932  1.088104 -0.632963
```

(continues on next page)

(continued from previous page)

```
2000-01-03 -0.585314 -0.275038 -0.937512
2000-01-04  0.632369 -1.249657  0.975593
2000-01-05  1.060617 -0.143682  0.218423
2000-01-06  3.050329  1.317933 -0.963725
2000-01-07 -0.539452 -0.771133  0.023751
2000-01-08  0.649464 -1.736427  0.197288
GROUP: /foo/bar
KEY: /foo/bar/bah
                   A         B         C
2000-01-01  1.334065  0.521036  0.930384
2000-01-02 -1.613932  1.088104 -0.632963
2000-01-03 -0.585314 -0.275038 -0.937512
2000-01-04  0.632369 -1.249657  0.975593
2000-01-05  1.060617 -0.143682  0.218423
2000-01-06  3.050329  1.317933 -0.963725
2000-01-07 -0.539452 -0.771133  0.023751
2000-01-08  0.649464 -1.736427  0.197288
```

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' (Array)]
```

Instead, use explicit string based keys:

```
In [386]: store['foo/bar/bah']
Out[386]:
                   A         B         C
2000-01-01  1.334065  0.521036  0.930384
2000-01-02 -1.613932  1.088104 -0.632963
2000-01-03 -0.585314 -0.275038 -0.937512
2000-01-04  0.632369 -1.249657  0.975593
2000-01-05  1.060617 -0.143682  0.218423
2000-01-06  3.050329  1.317933 -0.963725
2000-01-07 -0.539452 -0.771133  0.023751
2000-01-08  0.649464 -1.736427  0.197288
```

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

In [388]: df_mixed.loc[df_mixed.index[3:5],
   .....:              ['A', 'B', 'string', 'datetime64']] = np.nan
   .....: 

In [389]: store.append('df_mixed', df_mixed, min_itemsize={'values': 50})

In [390]: df_mixed1 = store.select('df_mixed')

In [391]: df_mixed1
Out[391]: 
          A         B         C  string  int  bool datetime64
0 -0.116008  0.743946 -0.398501  string    1  True 2001-01-02
1  0.592375 -0.533097 -0.677311  string    1  True 2001-01-02
2  0.476481 -0.140850 -0.874991  string    1  True 2001-01-02
3       NaN       NaN -1.167564     NaN    1  True        NaT
4       NaN       NaN -0.593353     NaN    1  True        NaT
5  0.852727  0.463819  0.146262  string    1  True 2001-01-02
6 -1.177365  0.793644 -0.131959  string    1  True 2001-01-02
7  1.236988  0.221252  0.089012  string    1  True 2001-01-02

In [392]: df_mixed1.dtypes.value_counts()
Out[392]: 
float64           2
float32           1
datetime64[ns]    1
bool              1
object            1
int64             1
dtype: int64

# we have provided a minimum string column size
In [393]: store.root.df_mixed.table
Out[393]: 
/df_mixed/table (Table(8,)) ''
  description := {
```

(continues on next page)

(continued from previous page)

```
  "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 MultiIndex DataFrames

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

```
In [394]: index = pd.MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
   .....:                              ['one', 'two', 'three']],
   .....:                      codes=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
   .....:                             [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   .....:                      names=['foo', 'bar'])
   .....:

In [395]: df_mi = pd.DataFrame(np.random.randn(10, 3), index=index,
   .....:                      columns=['A', 'B', 'C'])
   .....:

In [396]: df_mi
Out[396]:
                  A         B         C
foo bar
foo one    0.667450  0.169405 -1.358046
    two   -0.105563  0.492195  0.076693
    three  0.213685 -0.285283 -1.210529
bar one   -1.408386  0.941577 -0.342447
    two    0.222031  0.052607  2.093214
baz two    1.064908  1.778161 -0.913867
    three -0.030004 -0.399846 -1.234765
qux one    0.081323 -0.268494  0.168016
    two   -0.898283 -0.218499  1.408028
    three -1.267828 -0.689263  0.520995

In [397]: store.append('df_mi', df_mi)

In [398]: store.select('df_mi')
Out[398]:
                  A         B         C
foo bar
foo one    0.667450  0.169405 -1.358046
    two   -0.105563  0.492195  0.076693
    three  0.213685 -0.285283 -1.210529
bar one   -1.408386  0.941577 -0.342447
    two    0.222031  0.052607  2.093214
```

(continues on next page)

(continued from previous page)

```
baz two    1.064908  1.778161 -0.913867
    three -0.030004 -0.399846 -1.234765
qux one    0.081323 -0.268494  0.168016
    two   -0.898283 -0.218499  1.408028
    three -1.267828 -0.689263  0.520995

# the levels are automatically included as data columns
In [399]: store.select('df_mi', 'foo=bar')
Out[399]:
                A         B         C
foo bar
bar one -1.408386  0.941577 -0.342447
    two  0.222031  0.052607  2.093214
```

Note: The `index` keyword is reserved and cannot be use as a level name.

Querying

Querying a table

`select` and `delete` operations have an optional criterion that can be specified to select/delete only a subset of the data. This allows one to have a very large on-disk table and retrieve only a portion of the data.

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

- `index` and `columns` are supported indexers of `DataFrames`.
- if `data_columns` are specified, these can be used as additional indexers.
- level name in a MultiIndex, with default name `level_0`, `level_1`, … if not provided.

Valid comparison operators are:

`=, ==, !=, >, >=, <, <=`

Valid boolean expressions are combined with:

- `|` : or
- `&` : and
- `(` and `)` : for grouping

These rules are similar to how boolean expressions are used in pandas for indexing.

Note:

- = will be automatically expanded to the comparison operator ==
- ~ is the not operator, but can only be used in very limited circumstances
- If a list/tuple of expressions is passed they will be combined via `&`

The following are valid expressions:

- `'index >= date'`
- `"columns = ['A', 'D']"`

- `"columns in ['A', 'D']"`
- `'columns = A'`
- `'columns == A'`
- `"~(columns = ['A', 'B'])"`
- `'index > df.index[3] & string = "bar"'`
- `'(index > df.index[3] & index <= df.index[6]) | string = "bar"'`
- `"ts >= Timestamp('2012-02-01')"`
- `"major_axis>=20130101"`

The `indexers` are on the left-hand side of the sub-expression:

`columns`, `major_axis`, `ts`

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

In [401]: store.append('dfq', dfq, format='table', data_columns=True)
```

Use boolean expressions, with in-line function evaluation.

```
In [402]: store.select('dfq', "index>pd.Timestamp('20130104') & columns=['A', 'B']")
Out[402]:
                   A         B
2013-01-05 -1.083889  0.811865
2013-01-06 -0.402227  1.618922
2013-01-07  0.948196  0.183573
2013-01-08 -1.043530 -0.708145
2013-01-09  0.813949  1.508891
2013-01-10  1.176488 -1.246093
```

Use inline column reference.

```
In [403]: store.select('dfq', where="A>0 or C>0")
Out[403]:
                   A         B         C         D
2013-01-01  0.620028  0.159416 -0.263043 -0.639244
2013-01-04 -0.536722  1.005707  0.296917  0.139796
2013-01-05 -1.083889  0.811865  1.648435 -0.164377
2013-01-07  0.948196  0.183573  0.145277  0.308146
2013-01-08 -1.043530 -0.708145  1.430905 -0.850136
2013-01-09  0.813949  1.508891 -1.556154  0.187597
2013-01-10  1.176488 -1.246093 -0.002726 -0.444249
```

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 [404]: store.select('df', "columns=['A', 'B']")
Out[404]:
                   A         B
2000-01-01  1.334065  0.521036
2000-01-02 -1.613932  1.088104
2000-01-03 -0.585314 -0.275038
2000-01-04  0.632369 -1.249657
2000-01-05  1.060617 -0.143682
2000-01-06  3.050329  1.317933
2000-01-07 -0.539452 -0.771133
2000-01-08  0.649464 -1.736427
```

`start` and `stop` parameters can be specified to limit the total search space. These are in terms of the total number of rows in a table.

Note: `select` will raise a `ValueError` if the query expression has an unknown variable reference. Usually this means that you are trying to select on a column that is **not** a data_column.

`select` will raise a `SyntaxError` if the query expression is not valid.

Query timedelta64[ns]

You can store and query using the `timedelta64[ns]` type. Terms can be specified in the format: `<float>(<unit>)`, where float may be signed (and fractional), and unit can be `D,s,ms,us,ns` for the timedelta. Here's an example:

```
In [405]: from datetime import timedelta

In [406]: dftd = pd.DataFrame({'A': pd.Timestamp('20130101'),
   .....:                      'B': [pd.Timestamp('20130101') + timedelta(days=i,
   .....:                                                                 seconds=10)
   .....:                            for i in range(10)]})
   .....:

In [407]: dftd['C'] = dftd['A'] - dftd['B']

In [408]: dftd
Out[408]:
           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 [409]: store.append('dftd', dftd, data_columns=True)

In [410]: store.select('dftd', "C<'-3.5D'")
Out[410]:
           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
```

Query MultiIndex

Selecting from a `MultiIndex` can be achieved by using the name of the level.

```
In [411]: df_mi.index.names
Out[411]: FrozenList(['foo', 'bar'])

In [412]: store.select('df_mi', "foo=baz and bar=two")
Out[412]:
                A         B         C
foo bar
baz two  1.064908  1.778161 -0.913867
```

If the `MultiIndex` levels names are `None`, the levels are automatically made available via the `level_n` keyword with `n` the level of the `MultiIndex` you want to select from.

```
In [413]: index = pd.MultiIndex(
   .....:     levels=[["foo", "bar", "baz", "qux"], ["one", "two", "three"]],
   .....:     codes=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3], [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
   .....: )
   .....:

In [414]: df_mi_2 = pd.DataFrame(np.random.randn(10, 3),
   .....:                        index=index, columns=["A", "B", "C"])
   .....:

In [415]: df_mi_2
Out[415]:
                  A         B         C
foo one    0.856838  1.491776  0.001283
    two    0.701816 -1.097917  0.102588
    three  0.661740  0.443531  0.559313
bar one   -0.459055 -1.222598 -0.455304
    two   -0.781163  0.826204 -0.530057
baz two    0.296135  1.366810  1.073372
    three -0.994957  0.755314  2.119746
qux one   -2.628174 -0.089460 -0.133636
    two    0.337920 -0.634027  0.421107
    three  0.604303  1.053434  1.109090

In [416]: store.append("df_mi_2", df_mi_2)

# the levels are automatically included as data columns with keyword level_n
In [417]: store.select("df_mi_2", "level_0=foo and level_1=two")
Out[417]:
                A         B         C
foo two  0.701816 -1.097917  0.102588
```

Indexing

You can create/modify an index for a table with `create_table_index` after data is already in the table (after and `append/put` operation). Creating a table index is **highly** encouraged. This will speed your queries a great deal when you use a `select` with the indexed dimension as the `where`.

Note: Indexes are automagically created on the indexables and any data columns you specify. This behavior can be turned off by passing `index=False` to `append`.

```
# we have automagically already created an index (in the first section)
In [418]: i = store.root.df.table.cols.index.index

In [419]: i.optlevel, i.kind
Out[419]: (6, 'medium')

# change an index by passing new parameters
In [420]: store.create_table_index('df', optlevel=9, kind='full')

In [421]: i = store.root.df.table.cols.index.index
```

(continues on next page)

(continued from previous page)

```
In [422]: i.optlevel, i.kind
Out[422]: (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 [423]: df_1 = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [424]: df_2 = pd.DataFrame(np.random.randn(10, 2), columns=list('AB'))

In [425]: st = pd.HDFStore('appends.h5', mode='w')

In [426]: st.append('df', df_1, data_columns=['B'], index=False)

In [427]: st.append('df', df_2, data_columns=['B'], index=False)

In [428]: st.get_storer('df').table
Out[428]:
/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 [429]: st.create_table_index('df', columns=['B'], optlevel=9, kind='full')

In [430]: st.get_storer('df').table
Out[430]:
/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 [431]: 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 [432]: df_dc = df.copy()

In [433]: df_dc['string'] = 'foo'

In [434]: df_dc.loc[df_dc.index[4:6], 'string'] = np.nan

In [435]: df_dc.loc[df_dc.index[7:9], 'string'] = 'bar'

In [436]: df_dc['string2'] = 'cool'

In [437]: df_dc.loc[df_dc.index[1:3], ['B', 'C']] = 1.0

In [438]: df_dc
Out[438]:
                   A         B         C string string2
2000-01-01  1.334065  0.521036  0.930384    foo    cool
2000-01-02 -1.613932  1.000000  1.000000    foo    cool
2000-01-03 -0.585314  1.000000  1.000000    foo    cool
2000-01-04  0.632369 -1.249657  0.975593    foo    cool
2000-01-05  1.060617 -0.143682  0.218423    NaN    cool
2000-01-06  3.050329  1.317933 -0.963725    NaN    cool
2000-01-07 -0.539452 -0.771133  0.023751    foo    cool
2000-01-08  0.649464 -1.736427  0.197288    bar    cool

# on-disk operations
In [439]: store.append('df_dc', df_dc, data_columns=['B', 'C', 'string', 'string2'])

In [440]: store.select('df_dc', where='B > 0')
Out[440]:
                   A         B         C string string2
2000-01-01  1.334065  0.521036  0.930384    foo    cool
2000-01-02 -1.613932  1.000000  1.000000    foo    cool
2000-01-03 -0.585314  1.000000  1.000000    foo    cool
2000-01-06  3.050329  1.317933 -0.963725    NaN    cool

# getting creative
In [441]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')
Out[441]:
                   A         B         C string string2
2000-01-01  1.334065  0.521036  0.930384    foo    cool
2000-01-02 -1.613932  1.000000  1.000000    foo    cool
2000-01-03 -0.585314  1.000000  1.000000    foo    cool

# this is in-memory version of this type of selection
In [442]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]
Out[442]:
                   A         B         C string string2
2000-01-01  1.334065  0.521036  0.930384    foo    cool
2000-01-02 -1.613932  1.000000  1.000000    foo    cool
2000-01-03 -0.585314  1.000000  1.000000    foo    cool
```

(continues on next page)

(continued from previous page)

```
# we have automagically created this index and the B/C/string/string2
# columns are stored separately as ``PyTables`` columns
In [443]: store.root.df_dc.table
Out[443]:
/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

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 [444]: for df in store.select('df', chunksize=3):
   .....:     print(df)
   .....:
                   A         B         C
2000-01-01  1.334065  0.521036  0.930384
2000-01-02 -1.613932  1.088104 -0.632963
2000-01-03 -0.585314 -0.275038 -0.937512
                   A         B         C
2000-01-04  0.632369 -1.249657  0.975593
2000-01-05  1.060617 -0.143682  0.218423
2000-01-06  3.050329  1.317933 -0.963725
                   A         B         C
2000-01-07 -0.539452 -0.771133  0.023751
2000-01-08  0.649464 -1.736427  0.197288
```

Note: You can also use the iterator with `read_hdf` which will open, then automatically close the store when finished iterating.

```
for df in pd.read_hdf('store.h5', 'df', chunksize=3):
    print(df)
```

Note, that the chunksize keyword applies to the **source** rows. So if you are doing a query, then the chunksize will subdivide the total rows in the table and the query applied, returning an iterator on potentially unequal sized chunks.

Here is a recipe for generating a query and using it to create equal sized return chunks.

```
In [445]: dfeq = pd.DataFrame({'number': np.arange(1, 11)})

In [446]: dfeq
Out[446]:
   number
0       1
1       2
2       3
3       4
4       5
5       6
6       7
7       8
8       9
9      10

In [447]: store.append('dfeq', dfeq, data_columns=['number'])

In [448]: def chunks(l, n):
   .....:     return [l[i:i + n] for i in range(0, len(l), n)]
   .....:

In [449]: evens = [2, 4, 6, 8, 10]

In [450]: coordinates = store.select_as_coordinates('dfeq', 'number=evens')

In [451]: for c in chunks(coordinates, 2):
   .....:     print(store.select('dfeq', where=c))
   .....:
   number
1       2
3       4
   number
5       6
7       8
   number
9      10
```

Advanced queries

Select a single column

To retrieve a single indexable or data column, use the method `select_column`. This will, for example, enable you to get the index very quickly. These return a `Series` of the result, indexed by the row number. These do not currently accept the `where` selector.

```
In [452]: store.select_column('df_dc', 'index')
Out[452]:
0    2000-01-01
1    2000-01-02
2    2000-01-03
```

(continues on next page)