Filtering columns (usecols)

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

```
In [56]: data = 'a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz'

In [57]: pd.read_csv(StringIO(data))
Out[57]:
   a  b  c    d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

In [58]: pd.read_csv(StringIO(data), usecols=['b', 'd'])
Out[58]:
   b    d
0  2  foo
1  5  bar
2  8  baz

In [59]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
Out[59]:
   a  c    d
0  1  3  foo
1  4  6  bar
2  7  9  baz

In [60]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['A', 'C'])
Out[60]:
   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 [61]: pd.read_csv(StringIO(data), usecols=lambda x: x not in ['a', 'c'])
Out[61]:
   b    d
0  2  foo
1  5  bar
2  8  baz
```

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

Comments and empty lines

Ignoring line comments and empty lines

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

```
In [62]: data = ('\n'
   ....:         'a,b,c\n'
   ....:         '  \n'
```

(continues on next page)

(continued from previous page)

```
   ....:          '# commented line\n'
   ....:          '1,2,3\n'
   ....:          '\n'
   ....:          '4,5,6')
   ....:

In [63]: print(data)

a,b,c

# commented line
1,2,3

4,5,6

In [64]: pd.read_csv(StringIO(data), comment='#')
Out[64]:
   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 [65]: data = ('a,b,c\n'
   ....:          '\n'
   ....:          '1,2,3\n'
   ....:          '\n'
   ....:          '\n'
   ....:          '4,5,6')
   ....:

In [66]: pd.read_csv(StringIO(data), skip_blank_lines=False)
Out[66]:
     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 [67]: data = ('#comment\n'
   ....:          'a,b,c\n'
   ....:          'A,B,C\n'
   ....:          '1,2,3')
   ....:

In [68]: pd.read_csv(StringIO(data), comment='#', header=1)
Out[68]:
   A  B  C
0  1  2  3

In [69]: data = ('A,B,C\n'
```

```
   ....:          '#comment\n'
   ....:          'a,b,c\n'
   ....:          '1,2,3')
   ....:

In [70]: pd.read_csv(StringIO(data), comment='#', skiprows=2)
Out[70]:
   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 [71]: data = ('# empty\n'
   ....:         '# second empty line\n'
   ....:         '# third emptyline\n'
   ....:         'X,Y,Z\n'
   ....:         '1,2,3\n'
   ....:         'A,B,C\n'
   ....:         '1,2.,4.\n'
   ....:         '5.,NaN,10.0\n')
   ....:

In [72]: 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 [73]: pd.read_csv(StringIO(data), comment='#', skiprows=4, header=1)
Out[73]:
     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 [74]: 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 [75]: df = pd.read_csv('tmp.csv')

In [76]: df
```

(continues on next page)

(continued from previous page)

```
Out[76]:
         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 [77]: df = pd.read_csv('tmp.csv', comment='#')

In [78]: df
Out[78]:
         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 [79]: from io import BytesIO

In [80]: data = (b'word,length\n'
   ....:         b'Tr\xc3\xa4umen,7\n'
   ....:         b'Gr\xc3\xbc\xc3\x9fe,5')
   ....:

In [81]: data = data.decode('utf8').encode('latin-1')

In [82]: df = pd.read_csv(BytesIO(data), encoding='latin-1')

In [83]: df
Out[83]:
      word  length
0  Träumen       7
1    Grüße       5

In [84]: df['word'][1]
Out[84]: '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 [85]: data = ('a,b,c\n'
   ....:         '4,apple,bat,5.7\n'
   ....:         '8,orange,cow,10')
   ....:

In [86]: pd.read_csv(StringIO(data))
Out[86]:
        a    b     c
4   apple  bat   5.7
8  orange  cow  10.0
```

```
In [87]: data = ('index,a,b,c\n'
   ....:         '4,apple,bat,5.7\n'
   ....:         '8,orange,cow,10')
   ....:

In [88]: pd.read_csv(StringIO(data), index_col=0)
Out[88]:
            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 [89]: data = ('a,b,c\n'
   ....:         '4,apple,bat,\n'
   ....:         '8,orange,cow,')
   ....:

In [90]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [91]: pd.read_csv(StringIO(data))
Out[91]:
        a    b   c
4   apple  bat NaN
8  orange  cow NaN

In [92]: pd.read_csv(StringIO(data), index_col=False)
Out[92]:
   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 [93]: data = ('a,b,c\n'
   ....:         '4,apple,bat,\n'
   ....:         '8,orange,cow,')
   ....:

In [94]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [95]: pd.read_csv(StringIO(data), usecols=['b', 'c'])
Out[95]:
     b   c
4  bat NaN
8  cow NaN

In [96]: pd.read_csv(StringIO(data), usecols=['b', 'c'], index_col=0)
Out[96]:
     b   c
4  bat NaN
8  cow NaN
```

Date Handling

Specifying date columns

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

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

```
# Use a column as an index, and parse it as dates.
In [97]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)

In [98]: df
Out[98]:
            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 [99]: df.index
Out[99]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype=
↪'datetime64[ns]', name='date', freq=None)
```

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 [100]: 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 [101]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])

In [102]: df
Out[102]:
                  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 [103]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]],
   .....:                  keep_date_col=True)
   .....:

In [104]: df
Out[104]:
                  1_2                 1_3     0         1          2          3     4
0 1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  19990127   19:00:00   18:56:00  0.81
1 1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  19990127   20:00:00   19:56:00  0.01
2 1999-01-27 21:00:00 1999-01-27 20:56:00  KORD  19990127   21:00:00   20:56:00 -0.59
3 1999-01-27 21:00:00 1999-01-27 21:18:00  KORD  19990127   21:00:00   21:18:00 -0.99
4 1999-01-27 22:00:00 1999-01-27 21:56:00  KORD  19990127   22:00:00   21:56:00 -0.59
5 1999-01-27 23:00:00 1999-01-27 22:56:00  KORD  19990127   23:00:00   22:56:00 -0.59
```

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

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

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

In [106]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)

In [107]: df
Out[107]:
              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 [108]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [109]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   .....:                  index_col=0)  # index is the nominal column
   .....:

In [110]: df
Out[110]:
                                 actual     0     4
nominal
1999-01-27 19:00:00 1999-01-27 18:56:00  KORD  0.81
1999-01-27 20:00:00 1999-01-27 19:56:00  KORD  0.01
1999-01-27 21:00:00 1999-01-27 20:56:00  KORD -0.59
1999-01-27 21:00:00 1999-01-27 21:18:00  KORD -0.99
1999-01-27 22:00:00 1999-01-27 21:56:00  KORD -0.59
1999-01-27 23:00:00 1999-01-27 22:56:00  KORD -0.59
```

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

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

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

Date parsing functions

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

```
In [111]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
   .....:                  date_parser=pd.io.date_converters.parse_date_time)
   .....:

In [112]: df
Out[112]:
              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.

Parsing a CSV with mixed timezones

Pandas cannot natively represent a column or index with mixed timezones. If your CSV file contains columns with a mixture of timezones, the default result will be an object-dtype column with strings, even with `parse_dates`.

```
In [113]: content = """\
   .....: a
   .....: 2000-01-01T00:00:00+05:00
   .....: 2000-01-01T00:00:00+06:00"""
   .....:

In [114]: df = pd.read_csv(StringIO(content), parse_dates=['a'])

In [115]: df['a']
Out[115]:
0    2000-01-01 00:00:00+05:00
1    2000-01-01 00:00:00+06:00
Name: a, dtype: object
```

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

```
In [116]: df = pd.read_csv(StringIO(content), parse_dates=['a'],
   .....:                  date_parser=lambda col: pd.to_datetime(col, utc=True))
   .....:

In [117]: df['a']
Out[117]:
0   1999-12-31 19:00:00+00:00
```

(continues on next page)

(continued from previous page)

```
1   1999-12-31 18:00:00+00:00
Name: a, dtype: datetime64[ns, UTC]
```

Inferring datetime format

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

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

- "20111230"
- "2011/12/30"
- "20111230 00:00:00"
- "12/30/2011 00:00:00"
- "30/Dec/2011 00:00:00"
- "30/December/2011 00:00:00"

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

```
# Try to infer the format for the index column
In [118]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
   .....:                  infer_datetime_format=True)
   .....:

In [119]: df
Out[119]:
            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 [120]: print(open('tmp.csv').read())
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c

In [121]: pd.read_csv('tmp.csv', parse_dates=[0])
```

(continues on next page)

(continued from previous page)

```
Out[121]:
        date  value cat
0 2000-01-06      5   a
1 2000-02-06     10   b
2 2000-03-06     15   c

In [122]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])
Out[122]:
        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 [123]: val = '0.3066101993807095471566981359501369297504425048828125'

In [124]: data = 'a,b,c\n1,2,{0}'.format(val)

In [125]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                 float_precision=None)['c'][0] - float(val))
   .....:
Out[125]: 1.1102230246251565e-16

In [126]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                 float_precision='high')['c'][0] - float(val))
   .....:
Out[126]: 5.551115123125783e-17

In [127]: abs(pd.read_csv(StringIO(data), engine='c',
   .....:                 float_precision='round_trip')['c'][0] - float(val))
   .....:
Out[127]: 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 [128]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [129]: df = pd.read_csv('tmp.csv', sep='|')

In [130]: df
```

(continues on next page)

(continued from previous page)

```
Out[130]:
         ID      level  category
0  Patient1    123,000        x
1  Patient2     23,000        y
2  Patient3  1,234,018        z

In [131]: df.level.dtype
Out[131]: dtype('O')
```

The thousands keyword allows integers to be parsed correctly:

```
In [132]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [133]: df = pd.read_csv('tmp.csv', sep='|', thousands=',')

In [134]: df
Out[134]:
         ID    level category
0  Patient1   123000        x
1  Patient2    23000        y
2  Patient3  1234018        z

In [135]: df.level.dtype
Out[135]: dtype('int64')
```

NA values

To control which values are parsed as missing values (which are signified by NaN), specify a string in na_values. If you specify a list of strings, then all values in it are considered to be missing values. If you specify a number (a float, like 5.0 or an integer like 5), the corresponding equivalent values will also imply a missing value (in this case effectively [5.0, 5] are recognized as NaN).

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

The default NaN recognized values are ['-1.#IND', '1.#QNAN', '1.#IND', '-1.#QNAN', '#N/A N/A', '#N/A', 'N/A', 'n/a', 'NA', '<NA>', '#NA', 'NULL', 'null', 'NaN', '-NaN', 'nan', '-nan', ''].

Let us consider some examples:

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

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

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

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

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

Above, both NA and 0 as strings are NaN.

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

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

Infinity

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

Returning Series

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

```
In [136]: print(open('tmp.csv').read())
level
Patient1,123000
Patient2,23000
Patient3,1234018

In [137]: output = pd.read_csv('tmp.csv', squeeze=True)

In [138]: output
Out[138]:
Patient1     123000
Patient2      23000
Patient3    1234018
Name: level, dtype: int64

In [139]: type(output)
Out[139]: pandas.core.series.Series
```

Boolean values

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

```
In [140]: data = ('a,b,c\n'
   .....:         '1,Yes,2\n'
   .....:         '3,No,4')
   .....:

In [141]: print(data)
a,b,c
1,Yes,2
3,No,4

In [142]: pd.read_csv(StringIO(data))
Out[142]:
   a    b  c
0  1  Yes  2
1  3   No  4

In [143]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
```

(continues on next page)

(continued from previous page)

```
Out[143]:
   a      b  c
0  1   True  2
1  3  False  4
```

Handling "bad" lines

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

```
In [144]: data = ('a,b,c\n'
   .....:         '1,2,3\n'
   .....:         '4,5,6,7\n'
   .....:         '8,9,10')
   .....:

In [145]: pd.read_csv(StringIO(data))
---------------------------------------------------------------------------
ParserError                               Traceback (most recent call last)
<ipython-input-145-6388c394e6b8> in <module>
----> 1 pd.read_csv(StringIO(data))

/pandas-release/pandas/pandas/io/parsers.py in parser_f(filepath_or_buffer, sep,␣
↪delimiter, header, names, index_col, usecols, squeeze, prefix, mangle_dupe_cols,␣
↪dtype, engine, converters, true_values, false_values, skipinitialspace, skiprows,␣
↪skipfooter, nrows, na_values, keep_default_na, na_filter, verbose, skip_blank_lines,
↪ parse_dates, infer_datetime_format, keep_date_col, date_parser, dayfirst, cache_
↪dates, iterator, chunksize, compression, thousands, decimal, lineterminator,␣
↪quotechar, quoting, doublequote, escapechar, comment, encoding, dialect, error_bad_
↪lines, warn_bad_lines, delim_whitespace, low_memory, memory_map, float_precision)
    674         )
    675 
--> 676         return _read(filepath_or_buffer, kwds)
    677 
    678     parser_f.__name__ = name

/pandas-release/pandas/pandas/io/parsers.py in _read(filepath_or_buffer, kwds)
    452 
    453     try:
--> 454         data = parser.read(nrows)
    455     finally:
    456         parser.close()

/pandas-release/pandas/pandas/io/parsers.py in read(self, nrows)
   1131     def read(self, nrows=None):
   1132         nrows = _validate_integer("nrows", nrows)
-> 1133         ret = self._engine.read(nrows)
   1134 
   1135         # May alter columns / col_dict

/pandas-release/pandas/pandas/io/parsers.py in read(self, nrows)
   2035     def read(self, nrows=None):
   2036         try:
-> 2037             data = self._reader.read(nrows)
   2038         except StopIteration:
```

(continues on next page)

(continued from previous page)

```
   2039                 if self._first_chunk:

/pandas-release/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader.
↪read()

/pandas-release/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._
↪read_low_memory()

/pandas-release/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._
↪read_rows()

/pandas-release/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.TextReader._
↪tokenize_rows()

/pandas-release/pandas/pandas/_libs/parsers.pyx in pandas._libs.parsers.raise_parser_
↪error()

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

You can elect to skip bad lines:

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

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

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

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

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

Dialect

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

Suppose you had data with unenclosed quotes:

```
In [146]: 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 [147]: import csv

In [148]: dia = csv.excel()

In [149]: dia.quoting = csv.QUOTE_NONE

In [150]: pd.read_csv(StringIO(data), dialect=dia)
Out[150]:
       label1 label2 label3
index1     "a      c      e
index2      b      d      f
```

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

```
In [151]: data = 'a,b,c~1,2,3~4,5,6'

In [152]: pd.read_csv(StringIO(data), lineterminator='~')
Out[152]:
   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 [153]: data = 'a, b, c\n1, 2, 3\n4, 5, 6'

In [154]: print(data)
a, b, c
1, 2, 3
4, 5, 6

In [155]: pd.read_csv(StringIO(data), skipinitialspace=True)
Out[155]:
   a  b  c
0  1  2  3
1  4  5  6
```

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

Quoting and Escape Characters

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

```
In [156]: data = 'a,b\n"hello, \\"Bob\\", nice to see you",5'

In [157]: print(data)
a,b
"hello, \"Bob\", nice to see you",5

In [158]: pd.read_csv(StringIO(data), escapechar='\\')
Out[158]:
                               a  b
0  hello, "Bob", nice to see you  5
```

Files with fixed width columns

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

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

Consider a typical fixed-width data file:

```
In [159]: 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 [160]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [161]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)

In [162]: df
Out[162]: 
                 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 [163]: widths = [6, 14, 13, 10]

In [164]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [165]: df
Out[165]: 
        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.

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

In [167]: df
Out[167]:
                 1           2        3
0
id8141  360.242940  149.910199  11950.7
id1594  444.953632  166.985655  11788.4
id1849  364.136849  183.628767  11806.2
id1230  413.836124  184.375703  11916.8
id1948  502.953953  173.237159  12468.3
```

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

```
In [168]: pd.read_fwf('bar.csv', header=None, index_col=0).dtypes
Out[168]:
1    float64
2    float64
3    float64
dtype: object

In [169]: pd.read_fwf('bar.csv', header=None, dtype={2: 'object'}).dtypes
Out[169]:
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 [170]: 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 [171]: pd.read_csv('foo.csv')
Out[171]:
          A  B  C
20090101  a  1  2
```

(continues on next page)

(continued from previous page)

```
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 [172]: df = pd.read_csv('foo.csv', parse_dates=True)

In [173]: df.index
Out[173]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype=
↪'datetime64[ns]', freq=None)
```

Reading an index with a `MultiIndex`

Suppose you have data indexed by two columns:

```
In [174]: print(open('data/mindex_ex.csv').read())
year,indiv,zit,xit
1977,"A",1.2,.6
1977,"B",1.5,.5
1977,"C",1.7,.8
1978,"A",.2,.06
1978,"B",.7,.2
1978,"C",.8,.3
1978,"D",.9,.5
1978,"E",1.4,.9
1979,"C",.2,.15
1979,"D",.14,.05
1979,"E",.5,.15
1979,"F",1.2,.5
1979,"G",3.4,1.9
1979,"H",5.4,2.7
1979,"I",6.4,1.2
```

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

```
In [175]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0, 1])

In [176]: df
Out[176]:
             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
```

(continues on next page)

(continued from previous page)

```
     H      5.40  2.70
     I      6.40  1.20

In [177]: df.loc[1978]
Out[177]:
       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 [178]: from pandas._testing import makeCustomDataframe as mkdf

In [179]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)

In [180]: df.to_csv('mi.csv')

In [181]: 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 [182]: pd.read_csv('mi.csv', header=[0, 1, 2, 3], index_col=[0, 1])
Out[182]:
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
```

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

```
In [183]: print(open('mi2.csv').read())
,a,a,a,b,c,c
,q,r,s,t,u,v
```

(continues on next page)

(continued from previous page)

```
one,1,2,3,4,5,6
two,7,8,9,10,11,12

In [184]: pd.read_csv('mi2.csv', header=[0, 1], index_col=0)
Out[184]:
     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 [185]: print(open('tmp2.sv').read())
:0:1:2:3
0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.1356323710171934
1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-1.0442359662799567
2:-0.8618489633477999:-2.1045692188948086:-0.4949292740687813:1.071803807037338
3:0.7215551622443669:-0.7067711336300845:-1.0395749851146963:0.27185988554282986
4:-0.42497232978883753:0.567020349793672:0.27623201927771873:-1.0874006912859915
5:-0.6736897080883706:0.1136484096888855:-1.4784265524372235:0.5249876671147047
6:0.4047052186802365:0.5770459859204836:-1.7150020161146375:-1.0392684835147725
7:-0.3706468582364464:-1.1578922506419993:-1.344311812731667:0.8448851414248841
8:1.0757697837155533:-0.10904997528022223:1.6435630703622064:-1.4693879595399115
9:0.35702056413309086:-0.6746001037299882:-1.776903716971867:-0.9689138124473498


In [186]: pd.read_csv('tmp2.sv', sep=None, engine='python')
Out[186]:
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
8           8  1.075770 -0.109050  1.643563 -1.469388
9           9  0.357021 -0.674600 -1.776904 -0.968914
```

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 [187]: print(open('tmp.sv').read())
|0|1|2|3
0|0.4691122999071863|-0.2828633443286633|-1.5090585031735124|-1.1356323710171934
1|1.2121120250208506|-0.17321464905330858|0.11920871129693428|-1.0442359662799567
2|-0.8618489633477999|-2.1045692188948086|-0.4949292740687813|1.071803807037338
3|0.7215551622443669|-0.7067711336300845|-1.0395749851146963|0.27185988554282986
4|-0.42497232978883753|0.567020349793672|0.27623201927771873|-1.0874006912859915
5|-0.6736897080883706|0.1136484096888855|-1.4784265524372235|0.5249876671147047
6|0.4047052186802365|0.5770459859204836|-1.7150020161146375|-1.0392684835147725
7|-0.3706468582364464|-1.1578922506419993|-1.344311812731667|0.8448851414248841
8|1.0757697837155533|-0.10904997528022223|1.6435630703622064|-1.4693879595399115
9|0.35702056413309086|-0.6746001037299882|-1.776903716971867|-0.9689138124473498


In [188]: table = pd.read_csv('tmp.sv', sep='|')

In [189]: table
Out[189]: 
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
5           5 -0.673690  0.113648 -1.478427  0.524988
6           6  0.404705  0.577046 -1.715002 -1.039268
7           7 -0.370647 -1.157892 -1.344312  0.844885
8           8  1.075770 -0.109050  1.643563 -1.469388
9           9  0.357021 -0.674600 -1.776904 -0.968914
```

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

```
In [190]: reader = pd.read_csv('tmp.sv', sep='|', chunksize=4)

In [191]: reader
Out[191]: <pandas.io.parsers.TextFileReader at 0x7f5336de7550>

In [192]: 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
```

(continues on next page)

(continued from previous page)

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

In [194]: reader.get_chunk(5)
Out[194]:
   Unnamed: 0         0         1         2         3
0           0  0.469112 -0.282863 -1.509059 -1.135632
1           1  1.212112 -0.173215  0.119209 -1.044236
2           2 -0.861849 -2.104569 -0.494929  1.071804
3           3  0.721555 -0.706771 -1.039575  0.271860
4           4 -0.424972  0.567020  0.276232 -1.087401
```

Specifying the parser engine

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

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

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

Reading remote files

You can pass in a URL to a CSV file:

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

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

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

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

Writing out data

Writing to CSV format

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

- path_or_buf: A string path to the file to write or a file object. If a file object it must be opened with *newline=''*
- sep : Field delimiter for the output file (default ",")
- na_rep: A string representation of a missing value (default '')
- float_format: Format string for floating point numbers
- columns: Columns to write (default None)
- header: Whether to write out the column names (default True)
- index: whether to write row (index) names (default True)
- index_label: Column label(s) for index column(s) if desired. If None (default), and *header* and *index* are True, then the index names are used. (A sequence should be given if the DataFrame uses MultiIndex).
- mode : Python write mode, default 'w'
- encoding: a string representing the encoding to use if the contents are non-ASCII, for Python versions prior to 3
- line_terminator: Character sequence denoting line end (default *os.linesep*)
- quoting: Set quoting rules as in csv module (default csv.QUOTE_MINIMAL). Note that if you have set a *float_format* then floats are converted to strings and csv.QUOTE_NONNUMERIC will treat them as non-numeric
- quotechar: Character used to quote fields (default '"')
- doublequote: Control quoting of quotechar in fields (default True)
- escapechar: Character used to escape sep and quotechar when appropriate (default None)
- chunksize: Number of rows to write at a time
- date_format: Format string for datetime objects

Writing a formatted string

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

- buf default None, for example a StringIO object
- columns default None, which columns to write
- col_space default None, minimum width of each column.
- na_rep default NaN, representation of NA value
- formatters default None, a dictionary (by column) of functions each of which takes a single argument and returns a formatted string
- float_format default None, a function which takes a single (float) argument and returns a formatted string; to be applied to floats in the DataFrame.

- `sparsify` default True, set to False for a `DataFrame` with a hierarchical index to print every MultiIndex key at each row.
- `index_names` default True, will print the names of the indices
- `index` default True, will print the index (ie, row labels)
- `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.

2.1.2 JSON

Read and write `JSON` format files and strings.

Writing JSON

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

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

  Series:

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

  DataFrame:

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

  The format of the JSON string

`split`	dict like {index -> [index], columns -> [columns], data -> [values]}
`records`	list like [{column -> value}, … , {column -> value}]
`index`	dict like {index -> {column -> value}}
`columns`	dict like {column -> {index -> value}}
`values`	just the values array

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