(continued from previous page)

```
In [11]: tips = tips.drop('new_bill', axis=1)
```

Filtering

Filtering in Stata is done with an `if` clause on one or more columns.

```
list if total_bill > 10
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*.

```
In [12]: tips[tips['total_bill'] > 10].head()
Out[12]:
   total_bill   tip     sex smoker  day    time  size
0       14.99  1.01  Female     No  Sun  Dinner     2
2       19.01  3.50    Male     No  Sun  Dinner     3
3       21.68  3.31    Male     No  Sun  Dinner     2
4       22.59  3.61  Female     No  Sun  Dinner     4
5       23.29  4.71    Male     No  Sun  Dinner     4
```

If/then logic

In Stata, an `if` clause can also be used to create new columns.

```
generate bucket = "low" if total_bill < 10
replace bucket = "high" if total_bill >= 10
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [13]: tips['bucket'] = np.where(tips['total_bill'] < 10, 'low', 'high')

In [14]: tips.head()
Out[14]:
   total_bill   tip     sex smoker  day    time  size bucket
0       14.99  1.01  Female     No  Sun  Dinner     2   high
1        8.34  1.66    Male     No  Sun  Dinner     3    low
2       19.01  3.50    Male     No  Sun  Dinner     3   high
3       21.68  3.31    Male     No  Sun  Dinner     2   high
4       22.59  3.61  Female     No  Sun  Dinner     4   high
```

Date functionality

Stata provides a variety of functions to do operations on date/datetime columns.

```
generate date1 = mdy(1, 15, 2013)
generate date2 = date("Feb152015", "MDY")

generate date1_year = year(date1)
generate date2_month = month(date2)

* shift date to beginning of next month
```

(continues on next page)

(continued from previous page)

```
generate date1_next = mdy(month(date1) + 1, 1, year(date1)) if month(date1) != 12
replace date1_next = mdy(1, 1, year(date1) + 1) if month(date1) == 12
generate months_between = mofd(date2) - mofd(date1)

list date1 date2 date1_year date2_month date1_next months_between
```

The equivalent pandas operations are shown below. In addition to these functions, pandas supports other Time Series features not available in Stata (such as time zone handling and custom offsets) – see the *timeseries documentation* for more details.

```
In [15]: tips['date1'] = pd.Timestamp('2013-01-15')

In [16]: tips['date2'] = pd.Timestamp('2015-02-15')

In [17]: tips['date1_year'] = tips['date1'].dt.year

In [18]: tips['date2_month'] = tips['date2'].dt.month

In [19]: tips['date1_next'] = tips['date1'] + pd.offsets.MonthBegin()

In [20]: tips['months_between'] = (tips['date2'].dt.to_period('M')
   ....:                            - tips['date1'].dt.to_period('M'))
   ....:

In [21]: tips[['date1', 'date2', 'date1_year', 'date2_month', 'date1_next',
   ....:       'months_between']].head()
   ....:
Out[21]:
       date1      date2  date1_year  date2_month date1_next    months_between
0 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
1 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
2 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
3 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
4 2013-01-15 2015-02-15        2013            2 2013-02-01  <25 * MonthEnds>
```

Selection of columns

Stata provides keywords to select, drop, and rename columns.

```
keep sex total_bill tip

drop sex

rename total_bill total_bill_2
```

The same operations are expressed in pandas below. Note that in contrast to Stata, these operations do not happen in place. To make these changes persist, assign the operation back to a variable.

```
# keep
In [22]: tips[['sex', 'total_bill', 'tip']].head()
Out[22]:
      sex  total_bill   tip
0  Female       14.99  1.01
1    Male        8.34  1.66
```

(continues on next page)

(continued from previous page)

```
2    Male       19.01  3.50
3    Male       21.68  3.31
4  Female       22.59  3.61

# drop
In [23]: tips.drop('sex', axis=1).head()
Out[23]:
   total_bill   tip smoker  day    time  size
0       14.99  1.01     No  Sun  Dinner     2
1        8.34  1.66     No  Sun  Dinner     3
2       19.01  3.50     No  Sun  Dinner     3
3       21.68  3.31     No  Sun  Dinner     2
4       22.59  3.61     No  Sun  Dinner     4

# rename
In [24]: tips.rename(columns={'total_bill': 'total_bill_2'}).head()
Out[24]:
   total_bill_2   tip     sex smoker  day    time  size
0         14.99  1.01  Female     No  Sun  Dinner     2
1          8.34  1.66    Male     No  Sun  Dinner     3
2         19.01  3.50    Male     No  Sun  Dinner     3
3         21.68  3.31    Male     No  Sun  Dinner     2
4         22.59  3.61  Female     No  Sun  Dinner     4
```

Sorting by values

Sorting in Stata is accomplished via `sort`

```
sort sex total_bill
```

pandas objects have a *`DataFrame.sort_values()`* method, which takes a list of columns to sort by.

```
In [25]: tips = tips.sort_values(['sex', 'total_bill'])

In [26]: tips.head()
Out[26]:
     total_bill   tip     sex smoker   day    time  size
67         1.07  1.00  Female    Yes   Sat  Dinner     1
92         3.75  1.00  Female    Yes   Fri  Dinner     2
111        5.25  1.00  Female     No   Sat  Dinner     1
145        6.35  1.50  Female     No  Thur   Lunch     2
135        6.51  1.25  Female     No  Thur   Lunch     2
```

String processing

Finding length of string

Stata determines the length of a character string with the `strlen()` and `ustrlen()` functions for ASCII and Unicode strings, respectively.

```
generate strlen_time = strlen(time)
generate ustrlen_time = ustrlen(time)
```

Python determines the length of a character string with the `len` function. In Python 3, all strings are Unicode strings. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.

```
In [27]: tips['time'].str.len().head()
Out[27]:
67     6
92     6
111    6
145    5
135    5
Name: time, dtype: int64

In [28]: tips['time'].str.rstrip().str.len().head()
Out[28]:
67     6
92     6
111    6
145    5
135    5
Name: time, dtype: int64
```

Finding position of substring

Stata determines the position of a character in a string with the `strpos()` function. This takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```
generate str_position = strpos(sex, "ale")
```

Python determines the position of a character in a string with the `find()` function. `find` searches for the first position of the substring. If the substring is found, the function returns its position. Keep in mind that Python indexes are zero-based and the function will return -1 if it fails to find the substring.

```
In [29]: tips['sex'].str.find("ale").head()
Out[29]:
67     3
92     3
111    3
145    3
135    3
Name: sex, dtype: int64
```

Extracting substring by position

Stata extracts a substring from a string based on its position with the `substr()` function.

```
generate short_sex = substr(sex, 1, 1)
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [30]: tips['sex'].str[0:1].head()
Out[30]:
67     F
92     F
```

(continues on next page)

(continued from previous page)

```
111    F
145    F
135    F
Name: sex, dtype: object
```

Extracting nth word

The Stata `word()` function returns the nth word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate first_name = word(name, 1)
generate last_name = word(name, -1)
```

Python extracts a substring from a string based on its text by using regular expressions. There are much more powerful approaches, but this just shows a simple approach.

```
In [31]: firstlast = pd.DataFrame({'string': ['John Smith', 'Jane Cook']})

In [32]: firstlast['First_Name'] = firstlast['string'].str.split(" ", expand=True)[0]

In [33]: firstlast['Last_Name'] = firstlast['string'].str.rsplit(" ", expand=True)[0]

In [34]: firstlast
Out[34]:
       string First_Name Last_Name
0  John Smith       John      John
1   Jane Cook       Jane      Jane
```

Changing case

The Stata `strupper()`, `strlower()`, `strproper()`, `ustrupper()`, `ustrlower()`, and `ustrtitle()` functions change the case of ASCII and Unicode strings, respectively.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate upper = strupper(string)
generate lower = strlower(string)
generate title = strproper(string)
list
```

The equivalent Python functions are `upper`, `lower`, and `title`.

```
In [35]: firstlast = pd.DataFrame({'string': ['John Smith', 'Jane Cook']})

In [36]: firstlast['upper'] = firstlast['string'].str.upper()

In [37]: firstlast['lower'] = firstlast['string'].str.lower()

In [38]: firstlast['title'] = firstlast['string'].str.title()

In [39]: firstlast
Out[39]:
       string       upper       lower       title
0  John Smith  JOHN SMITH  john smith  John Smith
1   Jane Cook   JANE COOK   jane cook   Jane Cook
```

Merging

The following tables will be used in the merge examples

```
In [40]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
   ....:                     'value': np.random.randn(4)})
   ....:

In [41]: df1
Out[41]:
  key     value
0   A  0.469112
1   B -0.282863
2   C -1.509059
3   D -1.135632

In [42]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
   ....:                     'value': np.random.randn(4)})
   ....:

In [43]: df2
Out[43]:
  key     value
0   B  1.212112
1   D -0.173215
2   D  0.119209
3   E -1.044236
```

In Stata, to perform a merge, one data set must be in memory and the other must be referenced as a file name on disk. In contrast, Python must have both `DataFrames` already in memory.

By default, Stata performs an outer join, where all observations from both data sets are left in memory after the merge. One can keep only observations from the initial data set, the merged data set, or the intersection of the two by using the values created in the `_merge` variable.

```
* First create df2 and save to disk
clear
input str1 key
B
D
D
E
```

(continues on next page)

(continued from previous page)

```
end
generate value = rnormal()
save df2.dta

* Now create df1 in memory
clear
input str1 key
A
B
C
D
end
generate value = rnormal()

preserve

* Left join
merge 1:n key using df2.dta
keep if _merge == 1

* Right join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 2

* Inner join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 3

* Outer join
restore
merge 1:n key using df2.dta
```

pandas DataFrames have a `DataFrame.merge()` method, which provides similar functionality. Note that different join types are accomplished via the `how` keyword.

```
In [44]: inner_join = df1.merge(df2, on=['key'], how='inner')

In [45]: inner_join
Out[45]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209

In [46]: left_join = df1.merge(df2, on=['key'], how='left')

In [47]: left_join
Out[47]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

(continues on next page)

(continued from previous page)

```
In [48]: right_join = df1.merge(df2, on=['key'], how='right')

In [49]: right_join
Out[49]:
  key   value_x   value_y
0   B -0.282863  1.212112
1   D -1.135632 -0.173215
2   D -1.135632  0.119209
3   E       NaN -1.044236

In [50]: outer_join = df1.merge(df2, on=['key'], how='outer')

In [51]: outer_join
Out[51]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236
```

Missing data

Like Stata, pandas has a representation for missing data – the special float value `NaN` (not a number). Many of the semantics are the same; for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [52]: outer_join
Out[52]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E       NaN -1.044236

In [53]: outer_join['value_x'] + outer_join['value_y']
Out[53]:
0         NaN
1    0.929249
2         NaN
3   -1.308847
4   -1.016424
5         NaN
dtype: float64

In [54]: outer_join['value_x'].sum()
Out[54]: -3.5940742896293765
```

One difference is that missing data cannot be compared to its sentinel value. For example, in Stata you could do this to filter missing values.

```
* Keep missing values
list if value_x == .
* Keep non-missing values
list if value_x != .
```

This doesn't work in pandas. Instead, the `pd.isna()` or `pd.notna()` functions should be used for comparisons.

```
In [55]: outer_join[pd.isna(outer_join['value_x'])]
Out[55]:
  key  value_x   value_y
5   E      NaN -1.044236

In [56]: outer_join[pd.notna(outer_join['value_x'])]
Out[56]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059       NaN
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
```

Pandas also provides a variety of methods to work with missing data – some of which would be challenging to express in Stata. For example, there are methods to drop all rows with any missing values, replacing missing values with a specified value, like the mean, or forward filling from previous rows. See the *missing data documentation* for more.

```
# Drop rows with any missing value
In [57]: outer_join.dropna()
Out[57]:
  key   value_x   value_y
1   B -0.282863  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209

# Fill forwards
In [58]: outer_join.fillna(method='ffill')
Out[58]:
  key   value_x   value_y
0   A  0.469112       NaN
1   B -0.282863  1.212112
2   C -1.509059  1.212112
3   D -1.135632 -0.173215
4   D -1.135632  0.119209
5   E -1.135632 -1.044236

# Impute missing values with the mean
In [59]: outer_join['value_x'].fillna(outer_join['value_x'].mean())
Out[59]:
0    0.469112
1   -0.282863
2   -1.509059
3   -1.135632
4   -1.135632
5   -0.718815
Name: value_x, dtype: float64
```

GroupBy

Aggregation

Stata's `collapse` can be used to group by one or more key variables and compute aggregations on numeric columns.

```
collapse (sum) total_bill tip, by(sex smoker)
```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the *groupby documentation* for more details and examples.

```
In [60]: tips_summed = tips.groupby(['sex', 'smoker'])[['total_bill', 'tip']].sum()

In [61]: tips_summed.head()
Out[61]:
               total_bill     tip
sex    smoker
Female No          869.68  149.77
       Yes         527.27   96.74
Male   No         1725.75  302.00
       Yes        1217.07  183.07
```

Transformation

In Stata, if the group aggregations need to be used with the original data set, one would usually use `bysort` with `egen()`. For example, to subtract the mean for each observation by smoker group.

```
bysort sex smoker: egen group_bill = mean(total_bill)
generate adj_total_bill = total_bill - group_bill
```

pandas `groupby` provides a `transform` mechanism that allows these type of operations to be succinctly expressed in one operation.

```
In [62]: gb = tips.groupby('smoker')['total_bill']

In [63]: tips['adj_total_bill'] = tips['total_bill'] - gb.transform('mean')

In [64]: tips.head()
Out[64]:
     total_bill   tip     sex smoker   day    time  size  adj_total_bill
67         1.07  1.00  Female    Yes   Sat  Dinner     1      -17.686344
92         3.75  1.00  Female    Yes   Fri  Dinner     2      -15.006344
111        5.25  1.00  Female     No   Sat  Dinner     1      -11.938278
145        6.35  1.50  Female     No  Thur   Lunch     2      -10.838278
135        6.51  1.25  Female     No  Thur   Lunch     2      -10.678278
```

By group processing

In addition to aggregation, pandas `groupby` can be used to replicate most other `bysort` processing from Stata. For example, the following example lists the first observation in the current sort order by sex/smoker group.

```
bysort sex smoker: list if _n == 1
```

In pandas this would be written as:

```
In [65]: tips.groupby(['sex', 'smoker']).first()
Out[65]:
               total_bill   tip   day    time  size  adj_total_bill
sex    smoker
Female No            5.25  1.00   Sat  Dinner     1      -11.938278
       Yes           1.07  1.00   Sat  Dinner     1      -17.686344
Male   No            5.51  2.00  Thur   Lunch     2      -11.678278
       Yes           5.25  5.15   Sun  Dinner     2      -13.506344
```

Other considerations

Disk vs memory

Pandas and Stata both operate exclusively in memory. This means that the size of data able to be loaded in pandas is limited by your machine's memory. If out of core processing is needed, one possibility is the dask.dataframe library, which provides a subset of pandas functionality for an on-disk `DataFrame`.

1.4.8 Tutorials

This is a guide to many pandas tutorials, geared mainly for new users.

Internal guides

pandas' own *10 Minutes to pandas*.

More complex recipes are in the *Cookbook*.

A handy pandas cheat sheet.

Community guides

pandas Cookbook by Julia Evans

The goal of this 2015 cookbook (by Julia Evans) is to give you some concrete examples for getting started with pandas. These are examples with real-world data, and all the bugs and weirdness that entails. For the table of contents, see the pandas-cookbook GitHub repository.

Learn Pandas by Hernan Rojas

A set of lesson for new pandas users: https://bitbucket.org/hrojas/learn-pandas

Practical data analysis with Python

This guide is an introduction to the data analysis process using the Python data ecosystem and an interesting open dataset. There are four sections covering selected topics as munging data, aggregating data, visualizing data and time series.

Exercises for new users

Practice your skills with real data sets and exercises. For more resources, please visit the main repository.

Modern pandas

Tutorial series written in 2016 by Tom Augspurger. The source may be found in the GitHub repository TomAugspurger/effective-pandas.

- Modern Pandas
- Method Chaining
- Indexes
- Performance
- Tidy Data
- Visualization
- Timeseries

Excel charts with pandas, vincent and xlsxwriter

- Using Pandas and XlsxWriter to create Excel charts

Video tutorials

- Pandas From The Ground Up (2015) (2:24) GitHub repo
- Introduction Into Pandas (2016) (1:28) GitHub repo
- Pandas: .head() to .tail() (2016) (1:26) GitHub repo
- Data analysis in Python with pandas (2016-2018) GitHub repo and Jupyter Notebook
- Best practices with pandas (2018) GitHub repo and Jupyter Notebook

Various tutorials

- Wes McKinney's (pandas BDFL) blog
- Statistical analysis made easy in Python with SciPy and pandas DataFrames, by Randal Olson
- Statistical Data Analysis in Python, tutorial videos, by Christopher Fonnesbeck from SciPy 2013
- Financial analysis in Python, by Thomas Wiecki
- Intro to pandas data structures, by Greg Reda
- Pandas and Python: Top 10, by Manish Amde
- Pandas DataFrames Tutorial, by Karlijn Willems
- A concise tutorial with real life examples

CHAPTER

TWO

USER GUIDE

The User Guide covers all of pandas by topic area. Each of the subsections introduces a topic (such as "working with missing data"), and discusses how pandas approaches the problem, with many examples throughout.

Users brand-new to pandas should start with 10min.

Further information on any specific method can be obtained in the *API reference*.

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

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

Format Type	Data Description	Reader	Writer
text	CSV	*read_csv*	*to_csv*
text	Fixed-Width Text File	*read_fwf*	
text	JSON	*read_json*	*to_json*
text	HTML	*read_html*	*to_html*
text	Local clipboard	*read_clipboard*	*to_clipboard*
	MS Excel	*read_excel*	*to_excel*
binary	OpenDocument	*read_excel*	
binary	HDF5 Format	*read_hdf*	*to_hdf*
binary	Feather Format	*read_feather*	*to_feather*
binary	Parquet Format	*read_parquet*	*to_parquet*
binary	ORC Format	*read_orc*	
binary	Msgpack	*read_msgpack*	*to_msgpack*
binary	Stata	*read_stata*	*to_stata*
binary	SAS	*read_sas*	
binary	SPSS	*read_spss*	
binary	Python Pickle Format	*read_pickle*	*to_pickle*
SQL	SQL	*read_sql*	*to_sql*
SQL	Google BigQuery	*read_gbq*	*to_gbq*

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

Note: For examples that use the `StringIO` class, make sure you import it according to your Python version, i.e. `from StringIO import StringIO` for Python 2 and `from io import StringIO` for Python 3.

2.1.1 CSV & text files

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

Parsing options

`read_csv()` accepts the following common arguments:

Basic

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

sep [str, defaults to `','` for `read_csv()`, `\t` for `read_table()`] Delimiter to use. If sep is `None`, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\\r\\t'`.

delimiter [str, default `None`] Alternative argument name for sep.

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

Column and index locations and names

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

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

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

index_col [int, str, sequence of int / str, or False, default `None`] Column(s) to use as the row labels of the `DataFrame`, either given as string name or column index. If a sequence of int / str is given, a MultiIndex is used.

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

usecols [list-like or callable, default `None`] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in *names* or inferred from the document header row(s). For example, a valid list-like *usecols* parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`.

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

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

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

In [2]: from io import StringIO

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

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

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

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

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

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

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

General parsing configuration

dtype [Type name or dict of column -> type, default `None`] Data type for data or columns. E.g. `{'a': np.float64, 'b': np.int32}` (unsupported with `engine='python'`). Use *str* or *object* together with suitable `na_values` settings to preserve and not interpret dtype.

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

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

true_values [list, default `None`] Values to consider as `True`.

false_values [list, default `None`] Values to consider as `False`.

skipinitialspace [boolean, default `False`] Skip spaces after delimiter.

skiprows [list-like or integer, default `None`] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

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

```
In [6]: data = ('col1,col2,col3\n'
   ...:         'a,b,1\n'
   ...:         'a,b,2\n'
   ...:         'c,d,3')
   ...:

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

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

skipfooter [int, default `0`] Number of lines at bottom of file to skip (unsupported with engine='c').

nrows [int, default `None`] Number of rows of file to read. Useful for reading pieces of large files.

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

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

NA and missing data handling

na_values [scalar, str, list-like, or dict, default `None`] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. See *na values const* below for a list of the values interpreted as NaN by default.

keep_default_na [boolean, default `True`] Whether or not to include the default NaN values when parsing the data. Depending on whether *na_values* is passed in, the behavior is as follows:

- If *keep_default_na* is `True`, and *na_values* are specified, *na_values* is appended to the default NaN values used for parsing.
- If *keep_default_na* is `True`, and *na_values* are not specified, only the default NaN values are used for parsing.
- If *keep_default_na* is `False`, and *na_values* are specified, only the NaN values specified *na_values* are used for parsing.
- If *keep_default_na* is `False`, and *na_values* are not specified, no strings will be parsed as NaN.

Note that if *na_filter* is passed in as `False`, the *keep_default_na* and *na_values* parameters will be ignored.

na_filter [boolean, default `True`] Detect missing value markers (empty strings and the value of na_values). In data without any NAs, passing `na_filter=False` can improve the performance of reading a large file.

verbose [boolean, default `False`] Indicate number of NA values placed in non-numeric columns.

skip_blank_lines [boolean, default `True`] If `True`, skip over blank lines rather than interpreting as NaN values.

Datetime handling

parse_dates [boolean or list of ints or names or list of lists or dict, default `False`.]

- If `True` -> try parsing the index.
- If `[1, 2, 3]` -> try parsing columns 1, 2, 3 each as a separate date column.
- If `[[1, 3]]` -> combine columns 1 and 3 and parse as a single date column.
- If `{'foo': [1, 3]}` -> parse columns 1, 3 as date and call result 'foo'. A fast-path exists for iso8601-formatted dates.

infer_datetime_format [boolean, default `False`] If `True` and parse_dates is enabled for a column, attempt to infer the datetime format to speed up the processing.

keep_date_col [boolean, default `False`] If `True` and parse_dates specifies combining multiple columns then keep the original columns.

date_parser [function, default `None`] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. pandas will try to call date_parser in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by parse_dates) as arguments; 2) concatenate (row-wise) the string values from the columns defined by parse_dates into a single array and pass that; and 3) call date_parser once for each row using one or more strings (corresponding to the columns defined by parse_dates) as arguments.

dayfirst [boolean, default `False`] DD/MM format dates, international and European format.

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

New in version 0.25.0.

Iteration

iterator [boolean, default `False`] Return *TextFileReader* object for iteration or getting chunks with `get_chunk()`.

chunksize [int, default `None`] Return *TextFileReader* object for iteration. See *iterating and chunking* below.

Quoting, compression, and file format

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

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

thousands [str, default `None`] Thousands separator.

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

float_precision [string, default None] Specifies which converter the C engine should use for floating-point values. The options are `None` for the ordinary converter, `high` for the high-precision converter, and `round_trip` for the round-trip converter.

lineterminator [str (length 1), default None] Character to break file into lines. Only valid with C parser.

quotechar [str (length 1)] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

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

doublequote [boolean, default True] When quotechar is specified and quoting is not QUOTE_NONE, indicate whether or not to interpret two consecutive quotechar elements **inside** a field as a single quotechar element.

escapechar [str (length 1), default None] One-character string used to escape delimiter when quoting is QUOTE_NONE.

comment [str, default None] Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as skip_blank_lines=True), fully commented lines are ignored by the parameter *header* but not by *skiprows*. For example, if comment='#', parsing '#empty\na,b,c\n1,2,3' with *header=0* will result in 'a,b,c' being treated as the header.

encoding [str, default None] Encoding to use for UTF when reading/writing (e.g. 'utf-8'). List of Python standard encodings.

dialect [str or csv.Dialect instance, default None] If provided, this parameter will override values (default or not) for the following parameters: *delimiter*, *doublequote*, *escapechar*, *skipinitialspace*, *quotechar*, and *quoting*. If it is necessary to override values, a ParserWarning will be issued. See csv.Dialect documentation for more details.

Error handling

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

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

Specifying column data types

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

In [15]: df = pd.read_csv(StringIO(data),
   ....:                  dtype={'b': object, 'c': np.float64, 'd': 'Int64'})
   ....:

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

Fortunately, pandas offers more than one way to ensure that your column(s) contain only one `dtype`. If you're unfamiliar with these concepts, you can see *here* to learn more about dtypes, and *here* to learn more about `object` conversion in pandas.

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

```
In [17]: data = ("col_1\n"
   ....:         "1\n"
   ....:         "2\n"
   ....:         "'A'\n"
   ....:         "4.22")
   ....:

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

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

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

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

Specifying categorical dtype

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

```
In [31]: data = ('col1,col2,col3\n'
   ....:         'a,b,1\n'
   ....:         'a,b,2\n'
   ....:         'c,d,3')
   ....:

In [32]: pd.read_csv(StringIO(data))
Out[32]:
  col1 col2  col3
0    a    b     1
1    a    b     2
2    c    d     3

In [33]: pd.read_csv(StringIO(data)).dtypes
Out[33]:
col1    object
col2    object
col3     int64
dtype: object

In [34]: pd.read_csv(StringIO(data), dtype='category').dtypes
Out[34]:
col1    category
col2    category
col3    category
dtype: object
```

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

```
In [35]: pd.read_csv(StringIO(data), dtype={'col1': 'category'}).dtypes
Out[35]:
col1    category
col2      object
col3       int64
dtype: object
```

New in version 0.21.0.

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

```
In [36]: from pandas.api.types import CategoricalDtype

In [37]: dtype = CategoricalDtype(['d', 'c', 'b', 'a'], ordered=True)

In [38]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).dtypes
Out[38]:
col1    category
col2      object
col3       int64
dtype: object
```

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

```
In [39]: dtype = CategoricalDtype(['a', 'b', 'd'])  # No 'c'

In [40]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).col1
Out[40]:
0      a
1      a
2    NaN
Name: col1, dtype: category
Categories (3, object): [a, b, d]
```

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

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

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

```
In [41]: df = pd.read_csv(StringIO(data), dtype='category')

In [42]: df.dtypes
Out[42]:
col1    category
col2    category
col3    category
dtype: object

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

In [44]: df['col3'].cat.categories = pd.to_numeric(df['col3'].cat.categories)

In [45]: df['col3']
Out[45]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, int64): [1, 2, 3]
```

Naming and using columns

Handling column names

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

```
In [46]: data = ('a,b,c\n'
   ....:         '1,2,3\n'
   ....:         '4,5,6\n'
   ....:         '7,8,9')
   ....:

In [47]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [48]: pd.read_csv(StringIO(data))
Out[48]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9
```

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

```
In [49]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [50]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
Out[50]:
   foo  bar  baz
0    1    2    3
1    4    5    6
2    7    8    9

In [51]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)
Out[51]:
  foo bar baz
0   a   b   c
1   1   2   3
2   4   5   6
3   7   8   9
```

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

```
In [52]: data = ('skip this skip it\n'
   ....:         'a,b,c\n'
   ....:         '1,2,3\n'
   ....:         '4,5,6\n'
   ....:         '7,8,9')
   ....:
```

(continues on next page)

(continued from previous page)

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

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

Duplicate names parsing

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

```
In [54]: data = ('a,b,a\n'
   ....:         '0,1,2\n'
   ....:         '3,4,5')
   ....:

In [55]: pd.read_csv(StringIO(data))
Out[55]:
   a  b  a.1
0  0  1    2
1  3  4    5
```

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

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

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

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