

Data Loading, Storage, and File Formats

Part 2

Reading and Writing Data in Text Format

Part 2

Reading Text Files in Pieces

- When processing very large files or figuring out the right set of arguments to correctly process a large file, you may only want to read in a small piece of a file or iterate through smaller chunks of the file.

- Before we look at a large file, we make the pandas display settings more compact:

```
In [26]: pd.options.display.max_rows = 10
```

- Now we have:

```
In [27]: result = pd.read_csv('examples/ex6.csv')
```

```
In [28]: result
```

```
Out[28]:
```

	one	two	three	four	key
0	0.467976	-0.038649	-0.295344	-1.824726	L
1	-0.358893	1.404453	0.704965	-0.200638	B
2	-0.501840	0.659254	-0.421691	-0.057688	G
3	0.204886	1.074134	1.388361	-0.982404	R
4	0.354628	-0.133116	0.283763	-0.837063	Q
...
9995	2.311896	-0.417070	-1.409599	-0.515821	L
9996	-0.479893	-0.650419	0.745152	-0.646038	E
9997	0.523331	0.787112	0.486066	1.093156	K
9998	-0.362559	0.598894	-1.843201	0.887292	G
9999	-0.096376	-1.012999	-0.657431	-0.573315	O

10000 rows × 5 columns

- If you want to only read a small number of rows (avoiding reading the entire file), specify that with `nrows`:

```
In [29]: pd.read_csv('examples/ex6.csv', nrows=5)
```

```
Out[29]:
```

	one	two	three	four	key
0	0.467976	-0.038649	-0.295344	-1.824726	L
1	-0.358893	1.404453	0.704965	-0.200638	B
2	-0.501840	0.659254	-0.421691	-0.057688	G
3	0.204886	1.074134	1.388361	-0.982404	R
4	0.354628	-0.133116	0.283763	-0.837063	Q

- To read a file in pieces, specify a `chunksize` as a number of rows:

```
In [30]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)
```

```
In [31]: chunker
```

```
Out[31]: <pandas.io.parsers.TextFileReader at 0x7fc501bd8b70>
```

- The `TextFileReader` object returned by `read_csv` allows you to iterate over the parts of the file according to the `chunksize`.
- For example, we can iterate over `ex6.csv`, aggregating the value counts in the 'key' column like so:

```
In [32]: chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)
```

```
In [33]: tot = pd.Series([])
```

```
In [34]: for piece in chunker:  
         tot = tot.add(piece['key'].value_counts(), fill_value=0)
```

```
In [35]: tot = tot.sort_values(ascending=False)
```

```
In [36]: tot[:10]
```

```
Out[36]: E    368.0  
         X    364.0  
         L    346.0  
         O    343.0  
         Q    340.0  
         M    338.0  
         J    337.0  
         F    335.0  
         K    334.0  
         H    330.0  
         dtype: float64
```

- `TextFileReader` is also equipped with a `get_chunk` method that enables you to read pieces of an arbitrary size.

Writing Data to Text Format

- Data can also be exported to a delimited format.
- Let's consider one of the CSV files read before:

```
In [37]: data = pd.read_csv('examples/ex5.csv')
```

```
In [38]: data
```

```
Out[38]:
```

	something	a	b	c	d	message
0	one	1	2	3.0	4	NaN
1	two	5	6	NaN	8	world
2	three	9	10	11.0	12	foo

- Using DataFrame's `to_csv` method, we can write the data out to a comma-separated file:

```
In [39]: data.to_csv('examples/out.csv')
```

```
In [40]: !cat examples/out.csv
```

```
,something,a,b,c,d,message  
0,one,1,2,3.0,4,  
1,two,5,6,,8,world  
2,three,9,10,11.0,12,foo
```

- Other delimiters can be used, of course (writing to `sys.stdout` so it prints the text result to the console):

```
In [41]: import sys
```

```
In [42]: data.to_csv(sys.stdout, sep='|')
```

```
|something|a|b|c|d|message  
0|one|1|2|3.0|4|  
1|two|5|6|8|world  
2|three|9|10|11.0|12|foo
```

- Missing values appear as empty strings in the output.
- You might want to denote them by some other sentinel value:

```
In [43]: data.to_csv(sys.stdout, na_rep='NULL')  
    ,something,a,b,c,d,message  
0,one,1,2,3.0,4,NULL  
1,two,5,6,NULL,8,world  
2,three,9,10,11.0,12,foo
```

- With no other options specified, both the row and column labels are written.
- Both of these can be disabled:

```
In [44]: data.to_csv(sys.stdout, index=False, header=False)
```

```
one,1,2,3.0,4,  
two,5,6,,8,world  
three,9,10,11.0,12,foo
```

- You can also write only a subset of the columns, and in an order of your choosing:

```
In [45]: data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])
```

```
a,b,c  
1,2,3.0  
5,6,  
9,10,11.0
```

- Series also has a `to_csv` method:

```
In [46]: dates = pd.date_range('1/1/2000', periods=7)
```

```
In [47]: ts = pd.Series(np.arange(7), index=dates)
```

```
In [48]: ts.to_csv('examples/tseries.csv')
```

```
/home/joshua/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1: FutureWarning: The signature of `Series.to_csv` was aligned to that of `DataFrame.to_csv`, and argument 'header' will change its default value from False to True: please pass an explicit value to suppress this warning.  
      """Entry point for launching an IPython kernel.
```

```
In [49]: !cat examples/tseries.csv
```

```
2000-01-01,0  
2000-01-02,1  
2000-01-03,2  
2000-01-04,3  
2000-01-05,4  
2000-01-06,5  
2000-01-07,6
```

Working with Delimited Formats

- It's possible to load most forms of tabular data from disk using functions like `pandas.read_table`.
- In some cases, however, some manual processing may be necessary.
- It's not uncommon to receive a file with one or more malformed lines that trip up `read_table`.

- To illustrate the basic tools, consider a small CSV file:

```
In [50]: !cat examples/ex7.csv
```

```
"a","b","c"  
"1","2","3"  
"1","2","3"
```

- For any file with a single-character delimiter, you can use Python's built-in `csv` module.
- To use it, pass any open file or file-like object to `csv.reader`:

```
In [51]: import csv  
         f = open('examples/ex7.csv')  
  
         reader = csv.reader(f)
```

- Iterating through the reader like a file yields tuples of values:

```
In [52]: for line in reader:  
         print(line)  
  
         ['a', 'b', 'c']  
         ['1', '2', '3']  
         ['1', '2', '3']
```

- From there, it's up to you to do the wrangling necessary to put the data in the form that you need it.
- Let's take this step by step.
- First, we read the file into a list of lines:

```
In [53]: with open('examples/ex7.csv') as f:  
        lines = list(csv.reader(f))
```

- Then, we split the lines into the header line and the data lines:

```
In [54]: header, values = lines[0], lines[1:]
```

- Then we can create a dictionary of data columns using a dictionary comprehension and the expression `zip(*values)`, which transposes rows to columns:

```
In [55]: data_dict = {h: v for h, v in zip(header, zip(*values))}
```

```
In [56]: data_dict
```

```
Out[56]: {'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

- CSV files come in many different flavors.
- To define a new format with a different delimiter, string quoting convention, or line terminator, we define a simple subclass of `csv.Dialect`:

```
In [57]: class my_dialect(csv.Dialect):  
         lineterminator = '\n'  
         delimiter = ';'   
         quotechar = '"'   
         quoting = csv.QUOTE_MINIMAL
```

```
In [58]: reader = csv.reader(f, dialect=my_dialect)
```

- We can also give individual CSV dialect parameters as keywords to `csv.reader` without having to define a subclass:

```
In [59]: reader = csv.reader(f, delimiter='|')
```

- The possible options (attributes of `csv.Dialect`) and what they do can be found in the following table.

Argument	Description
<code>delimiter</code>	One-character string to separate fields; defaults to <code>' '</code> .
<code>lineterminator</code>	Line terminator for writing; defaults to <code>'\r\n'</code> . Reader ignores this and recognizes cross-platform line terminators.
<code>quotechar</code>	Quote character for fields with special characters (like a delimiter); default is <code>'</code> .
<code>quoting</code>	Quoting convention. Options include <code>csv.QUOTE_ALL</code> (quote all fields), <code>csv.QUOTE_MINIMAL</code> (only fields with special characters like the delimiter), <code>csv.QUOTE_NONNUMERIC</code> , and <code>csv.QUOTE_NONE</code> (no quoting). See Python's documentation for full details. Defaults to <code>QUOTE_MINIMAL</code> .
<code>skipinitialspace</code>	Ignore whitespace after each delimiter; default is <code>False</code> .
<code>doublequote</code>	How to handle quoting character inside a field; if <code>True</code> , it is doubled (see online documentation for full detail and behavior).
<code>escapechar</code>	String to escape the delimiter if <code>quoting</code> is set to <code>csv.QUOTE_NONE</code> ; disabled by default.

- For files with more complicated or fixed multicharacter delimiters, you will not be able to use the `csv` module.
- In those cases, you'll have to do the line splitting and other cleanup using string's `split` method or the regular expression method `re.split`.

- To write delimited files manually, you can use `csv.writer`.
- It accepts an open, writable file object and the same dialect and format options as `csv.reader`:

```
In [60]: with open('mydata.csv', 'w') as f:
writer = csv.writer(f, dialect=my_dialect)
writer.writerow(('one', 'two', 'three'))
writer.writerow(('1', '2', '3'))
writer.writerow(('4', '5', '6'))
writer.writerow(('7', '8', '9'))
```