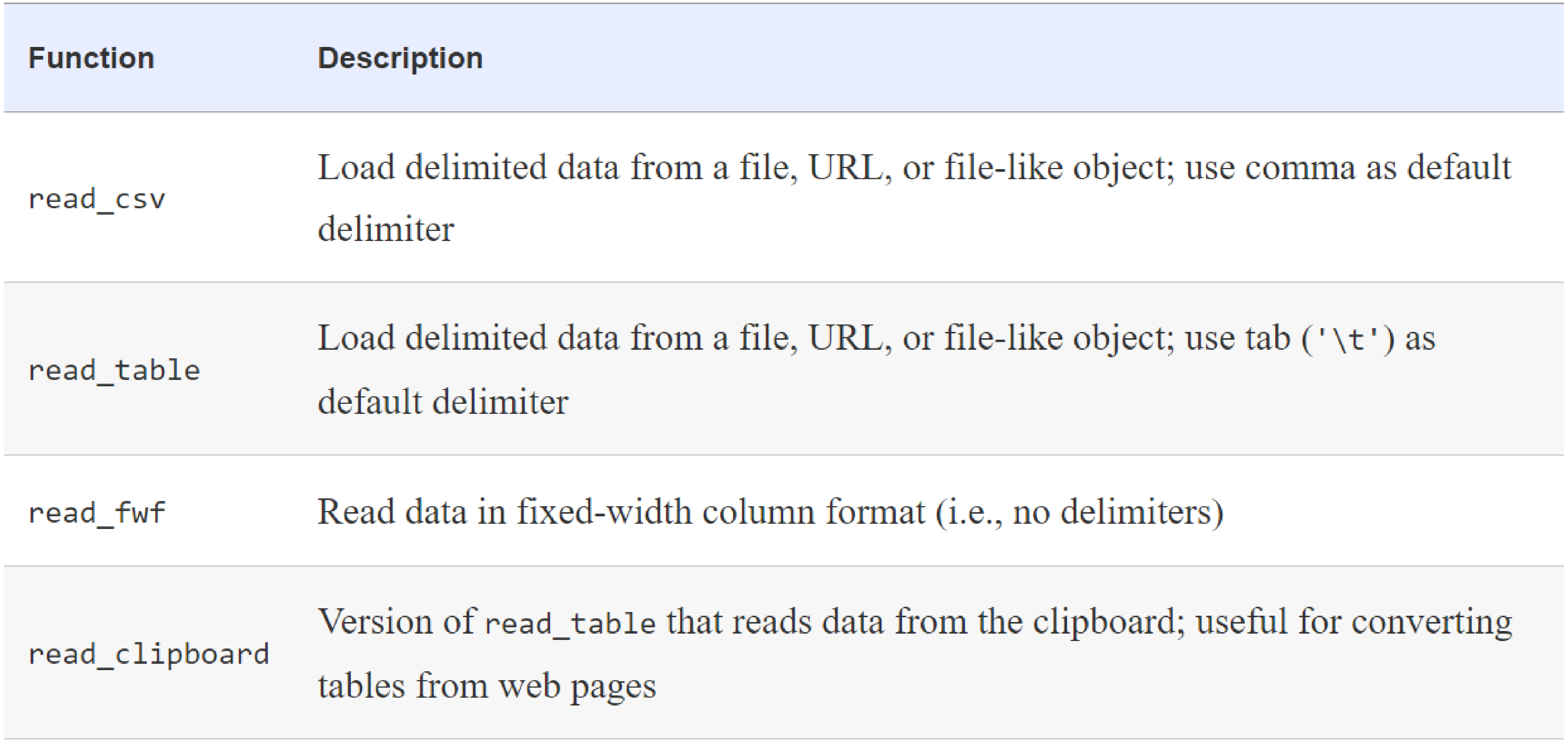
Data Loading, Storage, and File Formats

Part 1

Reading and Writing Data in Text Format

Part 1

Reading and Writing Data in Text Format

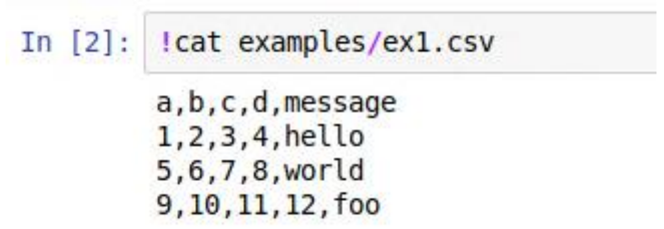


Function	Description
`read_csv`	Load delimited data from a file, URL, or file-like object; use comma as default delimiter
`read_table`	Load delimited data from a file, URL, or file-like object; use tab (`'\t'`) as default delimiter
`read_fwf`	Read data in fixed-width column format (i.e., no delimiters)
`read_clipboard`	Version of `read_table` that reads data from the clipboard; useful for converting tables from web pages

`read_excel`	Read tabular data from an Excel XLS or XLSX file
`read_hdf`	Read HDF5 files written by pandas
`read_html`	Read all tables found in the given HTML document
`read_json`	Read data from a JSON (JavaScript Object Notation) string representation
`read_msgpack`	Read pandas data encoded using the MessagePack binary format
`read_pickle`	Read an arbitrary object stored in Python pickle format

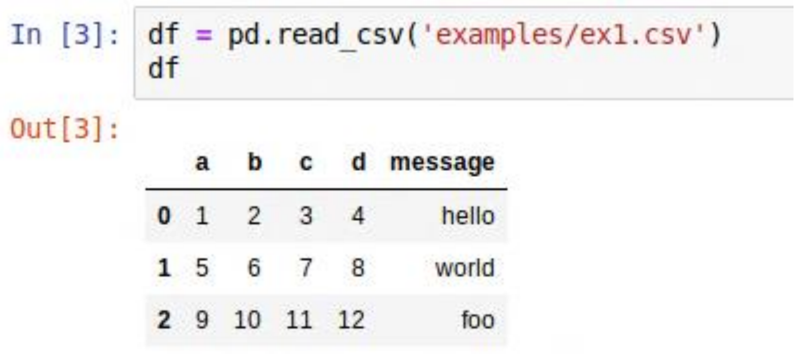
`read_sas`	Read a SAS dataset stored in one of the SAS system's custom storage formats
`read_sql`	Read the results of a SQL query (using SQLAlchemy) as a pandas DataFrame
`read_stata`	Read a dataset from Stata file format
`read_feather`	Read the Feather binary file format

- Let’s start with a small comma-separated (CSV) text file:



```
In [2]: !cat examples/ex1.csv
a,b,c,d,message
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

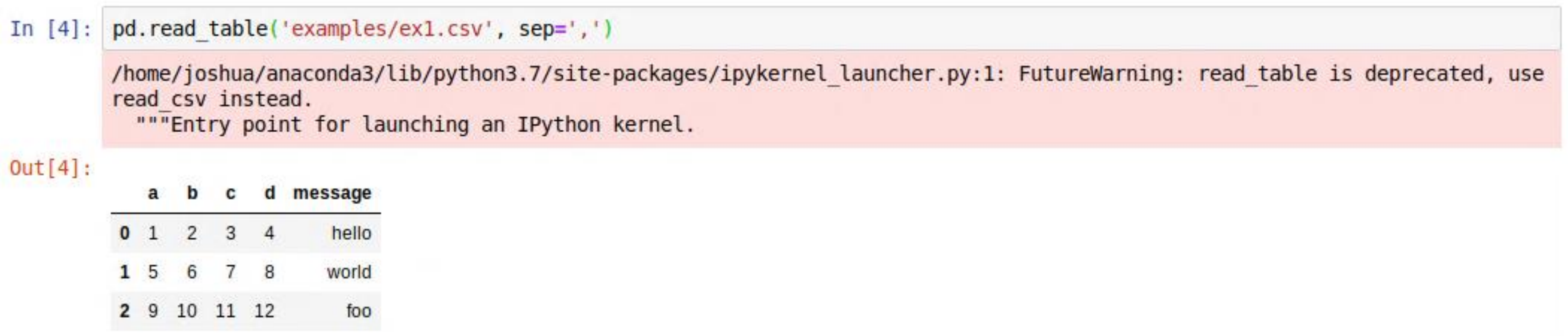
- Since this is comma-delimited, we can use `read_csv` to read it into a DataFrame:



```
In [3]: df = pd.read_csv('examples/ex1.csv')
        df

Out[3]:
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

- We could also have used `read_table` and specified the delimiter:



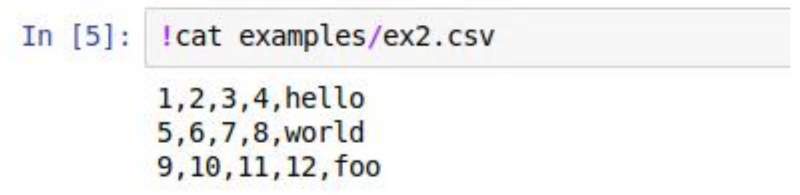
```
In [4]: pd.read_table('examples/ex1.csv', sep=',')
```

```
/home/joshua/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1: FutureWarning: read_table is deprecated, use
read_csv instead.
  """Entry point for launching an IPython kernel.
```

```
Out[4]:
```

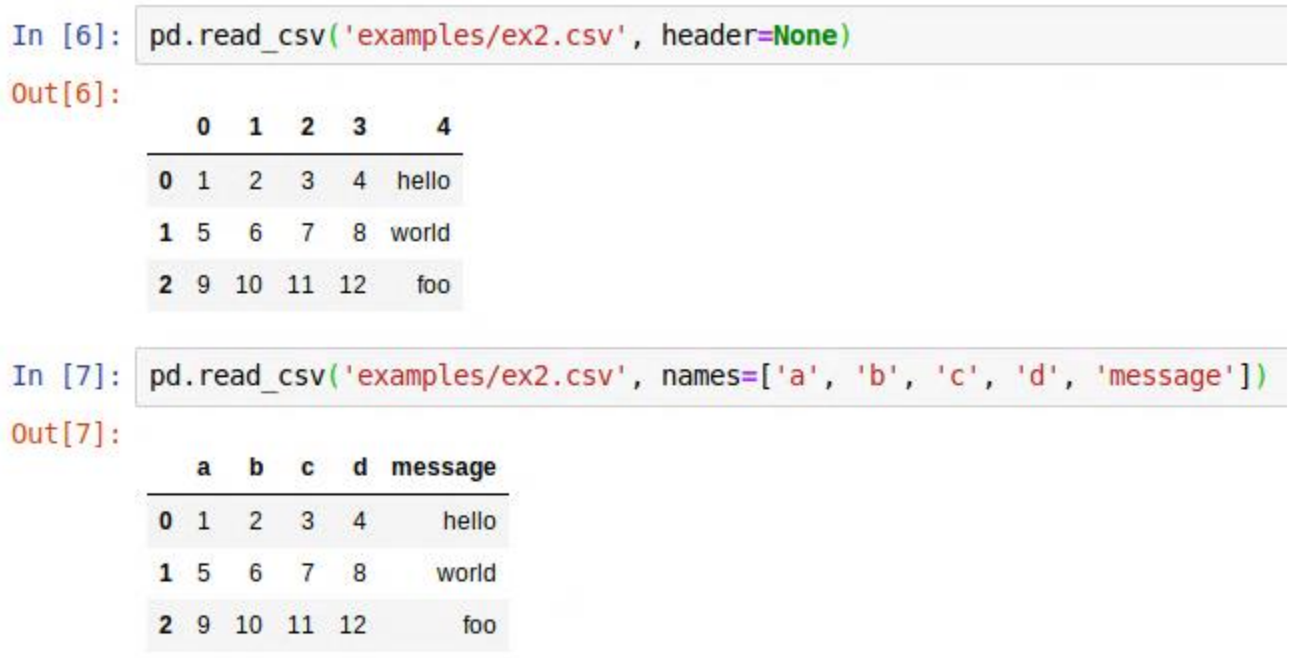
	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

- A file will not always have a header row.



```
In [5]: !cat examples/ex2.csv
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

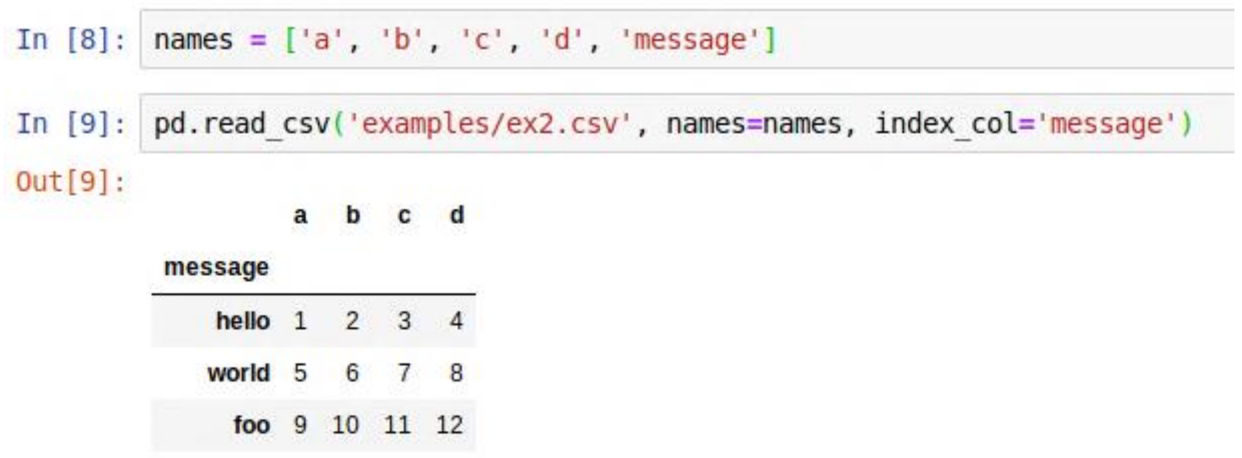
- To read this file, you have a couple of options.
- You can allow pandas to assign default column names, or you can specify names yourself:



```
In [6]: pd.read_csv('examples/ex2.csv', header=None)
Out[6]:
   0   1   2   3      4
0  1   2   3   4  hello
1  5   6   7   8  world
2  9  10  11  12    foo
```

```
In [7]: pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
Out[7]:
   a   b   c   d  message
0  1   2   3   4    hello
1  5   6   7   8    world
2  9  10  11  12      foo
```

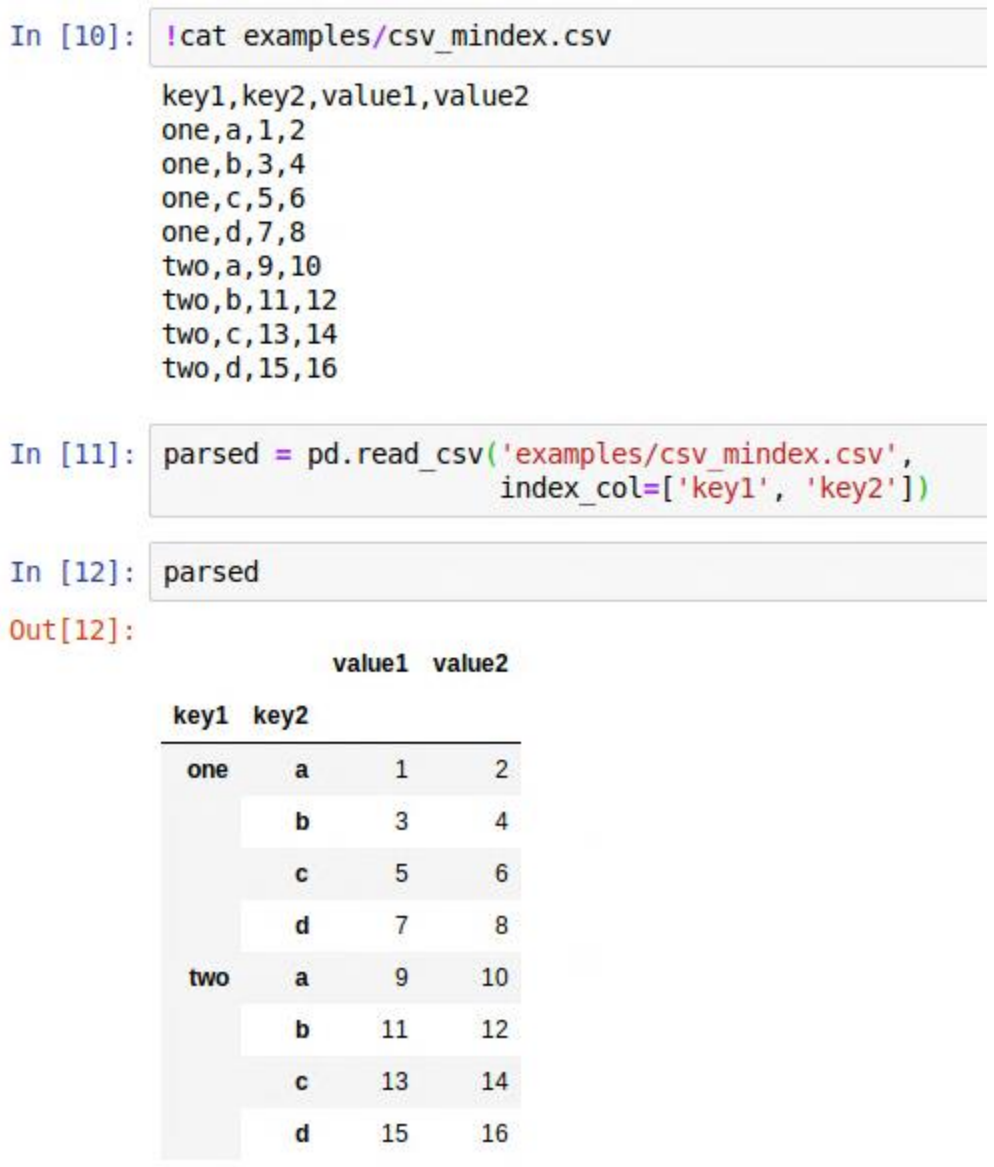
- Suppose you wanted the message column to be the index of the returned DataFrame.
- You can either indicate you want the column at index `4` or named `'message'` using the `index_col` argument:



```
In [8]: names = ['a', 'b', 'c', 'd', 'message']

In [9]: pd.read_csv('examples/ex2.csv', names=names, index_col='message')
Out[9]:
```

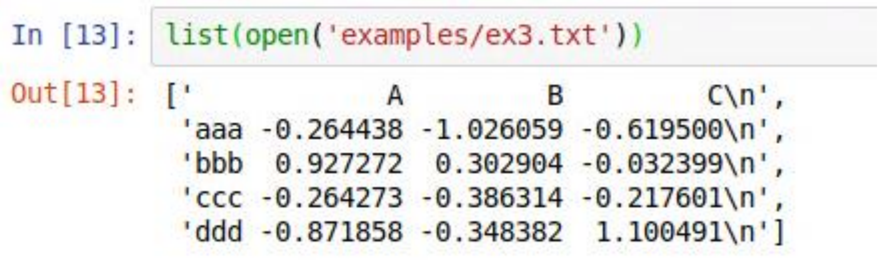
	a	b	c	d
message				
hello	1	2	3	4
world	5	6	7	8
foo	9	10	11	12

- In the event that you want to form a hierarchical index from multiple columns, pass a list of column numbers or names:



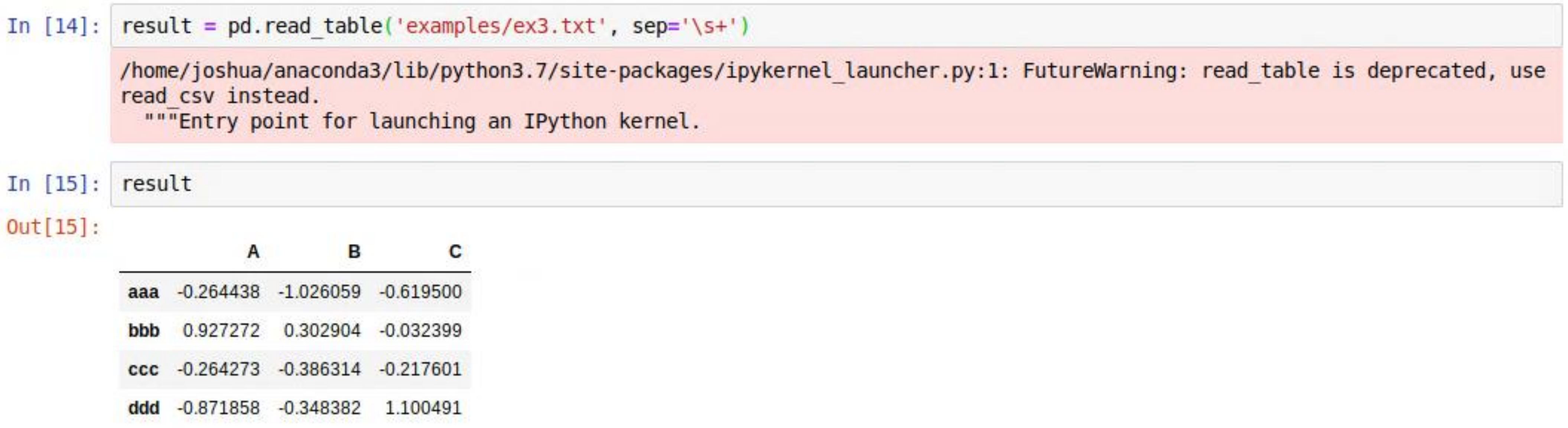
```
In [10]: !cat examples/csv_mindex.csv

key1,key2,value1,value2
one,a,1,2
one,b,3,4
one,c,5,6
one,d,7,8
two,a,9,10
two,b,11,12
two,c,13,14
two,d,15,16

In [11]: parsed = pd.read_csv('examples/csv_mindex.csv',
                              index_col=['key1', 'key2'])

In [12]: parsed

Out[12]:
                value1  value2
key1  key2
one   a              1       2
      b              3       4
      c              5       6
      d              7       8
two   a              9      10
      b             11      12
      c             13      14
      d             15      16
```

- In some cases, a table might not have a fixed delimiter, using whitespace or some other pattern to separate fields.
- Consider a text file that looks like this:



```
In [13]: list(open('examples/ex3.txt'))

Out[13]: ['            A         B         C\n',
          'aaa -0.264438 -1.026059 -0.619500\n',
          'bbb  0.927272  0.302904 -0.032399\n',
          'ccc -0.264273 -0.386314 -0.217601\n',
          'ddd -0.871858 -0.348382  1.100491\n']
```

- While you could do some munging by hand, the fields here are separated by a variable amount of whitespace.
- In these cases, you can pass a regular expression as a delimiter for `read_table`.
- This can be expressed by the regular expression `\s+`, so we have then:

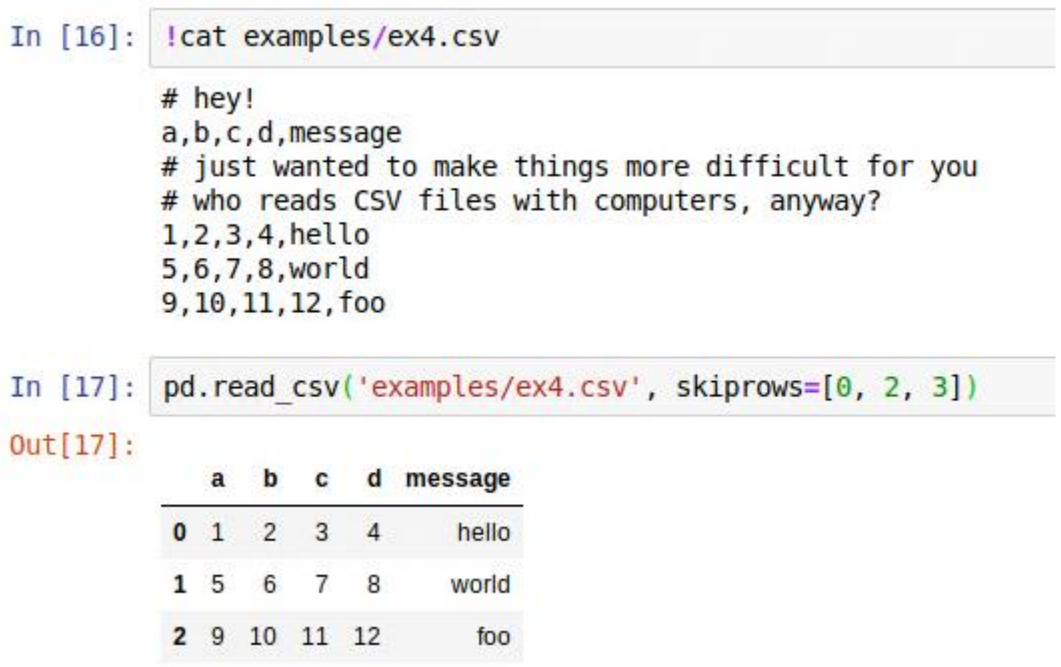


```
In [14]: result = pd.read_table('examples/ex3.txt', sep='\s+')

/home/joshua/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:1: FutureWarning: read_table is deprecated, use
read_csv instead.
  """Entry point for launching an IPython kernel.

In [15]: result
Out[15]:
```

	A	B	C
aaa	-0.264438	-1.026059	-0.619500
bbb	0.927272	0.302904	-0.032399
ccc	-0.264273	-0.386314	-0.217601
ddd	-0.871858	-0.348382	1.100491

- Because there was one fewer column name than the number of data rows, `read_table` infers that the first column should be the DataFrame’s index in this special case.

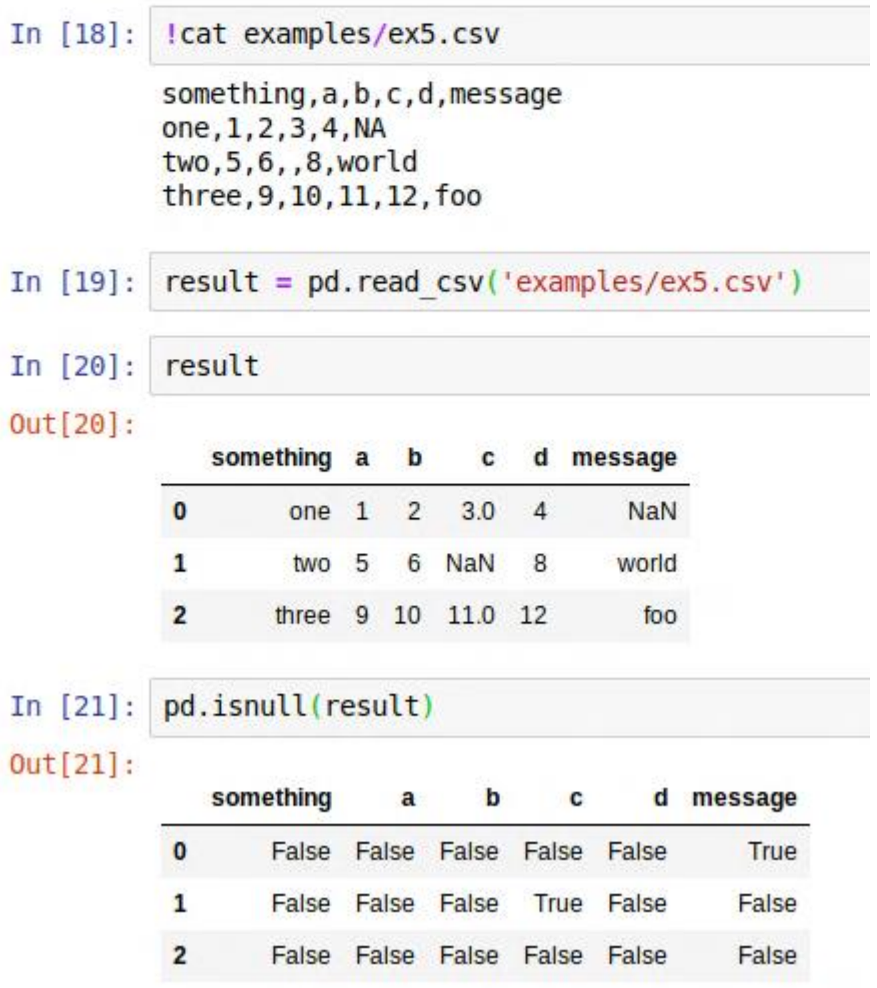
- The parser functions have many additional arguments to help you handle the wide variety of exception file formats that occur.
- For example, you can skip the first, third, and fourth rows of a file with `skiprows`:



```
In [16]: !cat examples/ex4.csv

# hey!
a,b,c,d,message
# just wanted to make things more difficult for you
# who reads CSV files with computers, anyway?
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo

In [17]: pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
Out[17]:
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

- Handling missing values is an important and frequently nuanced part of the file parsing process.
- Missing data is usually either not present (empty string) or marked by some *sentinel* value.
- By default, pandas uses a set of commonly occurring sentinels, such as NA and NULL:



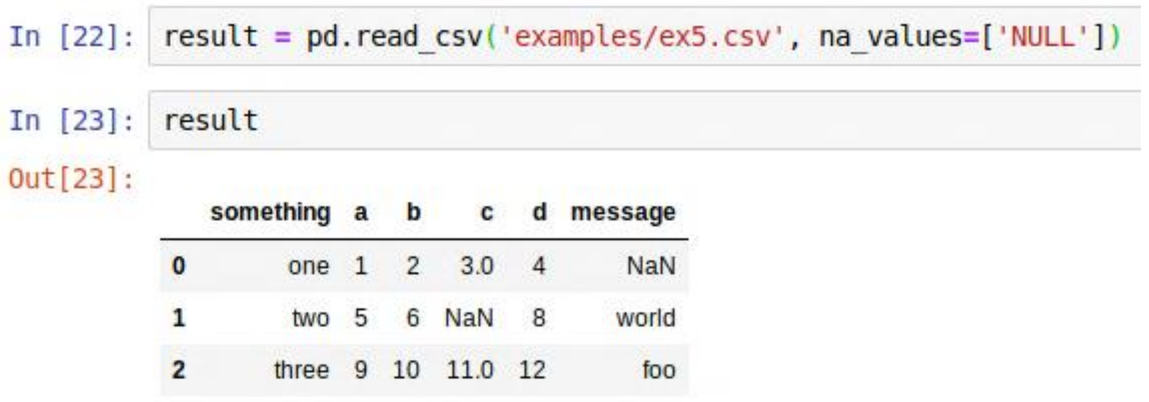
```
In [18]: !cat examples/ex5.csv
         something,a,b,c,d,message
         one,1,2,3,4,NA
         two,5,6,,8,world
         three,9,10,11,12,foo

In [19]: result = pd.read_csv('examples/ex5.csv')

In [20]: result
Out[20]:
```

	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

```
In [21]: pd.isnull(result)
Out[21]:
```

	something	a	b	c	d	message
0	False	False	False	False	False	True
1	False	False	False	True	False	False
2	False	False	False	False	False	False

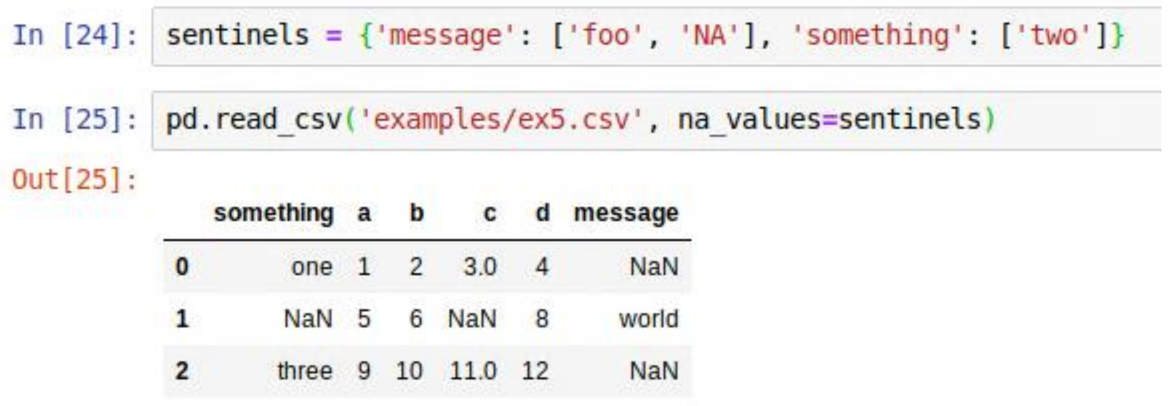
- The `na_values` option can take either a list or set of strings to consider missing values:



```
In [22]: result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])

In [23]: result
Out[23]:
```

	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

- Different NA sentinels can be specified for each column in a dict:



```
In [24]: sentinels = {'message': ['foo', 'NA'], 'something': ['two']}

In [25]: pd.read_csv('examples/ex5.csv', na_values=sentinels)
Out[25]:
  something  a   b     c   d  message
0       one  1   2   3.0   4      NaN
1       NaN  5   6   NaN   8    world
2     three  9  10  11.0  12      NaN
```