Data Loading, Storage, and File Formats

In this lecture we be focused on input and output with pandas objects, though there are of course numerous tools in other libraries to aid in this process.

Reading and Writing Data in Text Format

Python has become a beloved language for text and file munging due to its simple syntax for interacting with files, intuitive data structures, and convenient features like tuple packing and unpacking.

pandas features a number of functions for reading tabular data as a DataFrame object. Table 6-1 has a summary of all of them, though read_csv and read_table are likely the ones you'll use the most.

Table 6-1. Parsing functions in pandas

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 (that is, no delimiters)
`read_clipboard`	Version of `read_table` that reads data from the clipboard. Useful for converting tables from web pages

Overview of the mechanics of these functions, which are meant to convert text data into a DataFrame. The options for these functions fall into a few categories:

- Indexing: can treat one or more columns as the returned DataFrame, and whether to get column names from the file, the user, or not at all .

- Type inference and data conversion: this includes the user-defined value conversions and custom list of missing value markers.

- Datetime parsing: includes combining capability, including combining date and time information spread over multiple columns into a single column in the result.

- Iterating: support for iterating over chunks of very large files.

- Unclean data issues: skipping rows or a footer, comments, or other minor things like numeric data with thousands separated by commas.

Type inference

Type inference is one of the more important features of these functions; that means you don't have to specify which columns are numeric, integer, boolean, or string. Handling dates and other custom types requires a bit more effort, though. Let's start with a small comma-separated (CSV) text file:

In [1]:

```
!cat 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 [4]:

```
import pandas as pd
df = pd.read_csv('ex1.csv')
df
```

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

We could also have used read_table and specifying the delimiter:

In [5]:

```
pd.read_table('ex1.csv', sep=',')
```

```
/home/naz/.local/lib/python3.6/site-packages/ipykernel_launcher.py:1: FutureWarning: read_table is deprecated, use read_csv instead.
  """Entry point for launching an IPython kernel.
```

Out[5]:

	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. Consider this file:

In [6]:

```
!cat ex2.csv
```

```
1,2,3,4,hello
5,6,7,8,world
9,10,11,12,foo
```

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

In [7]:

```
pd.read_csv('ex2.csv', header=None)
```

Out[7]:

	0	1	2	3	4
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

In [8]:

```
pd.read_csv('ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
```

Out[8]:

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

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

In [10]:

```
!cat 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('csv_mindex.csv', index_col=['key1', 'key2'])
parsed
```

Out[11]:

		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. In these cases, you can pass a regular expression as a delimiter for read_table . Consider a text file that looks like this:

In [12]:

```
list(open('ex3.txt'))
```

Out[12]:

```
['            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, in this case fields are separated by a variable amount of whitespace. This can be expressed by the regular expression \s+ , so we have then:

In [14]:

```
result = pd.read_table('ex3.txt', sep='\s+')
result
```

```
/home/naz/.local/lib/python3.6/site-packages/ipykernel_launcher.py:1: FutureWarning: read_table is deprecated, use read_csv instead.
  """Entry point for launching an IPython kernel.
```

Out[14]:

	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

The parser functions have many additional arguments to help you handle the wide variety of exception file formats that occur (see Table 6-2). For

example, you can skip the first, third, and fourth rows of a file with skiprows :

In [15]:

```
!cat 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 [16]:

```
pd.read_csv('ex4.csv', skiprows=[0, 2, 3])
```

Out[16]:

	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 , -1.#IND , and NULL :

In [17]:

```
!cat 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 [18]:

```
result = pd.read_csv('ex5.csv')
result
```

Out[18]:

	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 [19]:

```
pd.isnull(result)
```

Out[19]:

	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 [21]:

```
result = pd.read_csv('ex5.csv', na_values=['NULL'])
```

```
result
```

Out[21]:

	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

Table 6-2. read_csv /read_table function arguments

Argument	Description
`path`	String indicating filesystem location, URL, or file-like object
`sep` or `delimiter`	Character sequence or regular expression to use to split fields in each row
`header`	Row number to use as column names. Defaults to 0 (first row), but should be None if there is no header row
`index_col`	Column numbers or names to use as the row index in the result. Can be a single name/number or a list of them for a hierarchical index
`names`	List of column names for result, combine with `header=None`
`skiprows`	Number of rows at beginning of file to ignore or list of row numbers (starting from 0) to skip
`na_values`	Sequence of values to replace with NA
`comment`	Character or characters to split comments off the end of lines
`parse_dates`	Attempt to parse data to datetime; False by default. If True, will attempt to parse all columns. Otherwise can specify a list of column numbers or name to parse. If element of list is tuple or list, will combine multiple columns together and parse to date (for example if date/time split across two columns)
`keep_date_col`	If joining columns to parse date, drop the joined columns. Default True
`converters`	Dict containing column number of name mapping to functions. For example `{'foo': f}` would apply the function `f` to all values in the `'foo'` column
`dayfirst`	When parsing potentially ambiguous dates, treat as international format (e.g. 7/6/2012 -> June 7, 2012). Default False
`date_parser`	Function to use to parse dates
`nrows`	Number of rows to read from beginning of file
`iterator`	Return a TextParser object for reading file piecemeal
`chunksize`	For iteration, size of file chunks
`skip_footer`	Number of lines to ignore at end of file
`verbose`	Print various parser output information, like the number of missing values placed in non-numeric columns
`encoding`	Text encoding for unicode. For example `'utf-8'` for UTF-8 encoded text
`squeeze`	If the parsed data only contains one column return a Series
`thousands`	Separator for thousands, e.g. `','` or `'.'`

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.

In [22]:

```
result = pd.read_csv('ex6.csv')
result
```

Out[22]:

	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

	one	two	three	four	key
3	0.204886	1.074134	1.388361	-0.982404	R
4	0.354628	-0.133116	0.283763	-0.837063	Q
5	1.817480	0.742273	0.419395	-2.251035	Q
6	-0.776764	0.935518	-0.332872	-1.875641	U
7	-0.913135	1.530624	-0.572657	0.477252	K
8	0.358480	-0.497572	-0.367016	0.507702	S
9	-1.740877	-1.160417	-1.637830	2.172201	G
10	0.240564	-0.328249	1.252155	1.072796	8
11	0.764018	1.165476	-0.639544	1.495258	R
12	0.571035	-0.310537	0.582437	-0.298765	1
13	2.317658	0.430710	-1.334216	0.199679	P
14	1.547771	-1.119753	-2.277634	0.329586	J
15	-1.310608	0.401719	-1.000987	1.156708	E
16	-0.088496	0.634712	0.153324	0.415335	B
17	-0.018663	-0.247487	-1.446522	0.750938	A
18	-0.070127	-1.579097	0.120892	0.671432	F
19	-0.194678	-0.492039	2.359605	0.319810	H
20	-0.248618	0.868707	-0.492226	-0.717959	W
21	-1.091549	-0.867110	-0.647760	-0.832562	C
22	0.641404	-0.138822	-0.621963	-0.284839	C
23	1.216408	0.992687	0.165162	-0.069619	V
24	-0.564474	0.792832	0.747053	0.571675	I
25	1.759879	-0.515666	-0.230481	1.362317	S
26	0.126266	0.309281	0.382820	-0.239199	L
27	1.334360	-0.100152	-0.840731	-0.643967	6
28	-0.737620	0.278087	-0.053235	-0.950972	J
29	-1.148486	-0.986292	-0.144963	0.124362	Y
...	...	...	...	...	...
9970	0.633495	-0.186524	0.927627	0.143164	4
9971	0.308636	-0.112857	0.762842	-1.072977	1
9972	-1.627051	-0.978151	0.154745	-1.229037	Z
9973	0.314847	0.097989	0.199608	0.955193	P
9974	1.666907	0.992005	0.496128	-0.686391	S
9975	0.010603	0.708540	-1.258711	0.226541	K
9976	0.118693	-0.714455	-0.501342	-0.254764	K
9977	0.302616	-2.011527	-0.628085	0.768827	H
9978	-0.098572	1.769086	-0.215027	-0.053076	A
9979	-0.019058	1.964994	0.738538	-0.883776	F
9980	-0.595349	0.001781	-1.423355	-1.458477	M
9981	1.392170	-1.396560	-1.425306	-0.847535	H
9982	-0.896029	-0.152287	1.924483	0.365184	6
9983	-2.274642	-0.901874	1.500352	0.996541	N
9984	-0.301898	1.019906	1.102160	2.624526	I
9985	-2.548389	-0.585374	1.496201	-0.718815	D
9986	-0.064588	0.759292	-1.568415	-0.420933	E
9987	-0.143365	-1.111760	-1.815581	0.435274	2
9988	-0.070412	-1.055921	0.338017	-0.440763	X
9989	0.649148	0.994273	-1.384227	0.485120	Q
9990	-0.370769	0.404356	-1.051628	-1.050899	8
9991	-0.409980	0.155627	-0.818990	1.277350	W
9992	0.301214	-1.111203	0.668258	0.671922	A
9993	1.821117	0.416445	0.173874	0.505118	X
9994	0.068804	1.322759	0.802346	0.223618	H
9995	2.311896	-0.417070	-1.409599	-0.515821	L

	one	two	three	four	key
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	0

10000 rows × 5 columns

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

In [23]:

```
pd.read_csv('ex6.csv', nrows=5)
```

Out[23]:

	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

Writing Data Out to Text Format

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

In [26]:

```
data = pd.read_csv('ex5.csv')
data
```

Out[26]:

	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 [27]:

```
data.to_csv('out.csv')
```

In [28]:

```
!cat 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 just prints the text result):

In [30]:

```
import sys
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 [32]:

```
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 [33]:

```
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 [35]:

```
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 [39]:

```
from pandas import Series
import numpy as np
dates = pd.date_range('1/1/2000', periods=7)
ts = Series(np.arange(7), index=dates)
ts.to_csv('tseries.csv')
```

/home/naz/.local/lib/python3.6/site-packages/ipykernel_launcher.py:5: 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.
  """

In [40]:

```
!cat 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

With a bit of wrangling (no header, first column as index), you can read a CSV version of a Series with read_csv , but there is also a from_csv convenience method that makes it a bit simpler:

In [41]:

```
Series.from_csv('tseries.csv', parse_dates=True)
```

/usr/local/lib/python3.6/dist-packages/pandas/core/series.py:4141: FutureWarning: from_csv is deprecated. Please use read_csv(...) instead. Note th

at some of the default arguments are different, so please refer to the documentation for from_csv when changing your function calls
infer_datetime_format=infer_datetime_format)

Out[41]:

```
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
dtype: int64
```

Manually Working with Delimited Formats

Most forms of tabular data can be loaded from disk using functions like pan das.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 [42]:

```
!cat ex7.csv
```

```
"a","b","c"
"1","2","3"
"1","2","3","4"
```

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 [44]:

```
import csv
f = open('ex7.csv')
reader = csv.reader(f)
for line in reader:
    print(line)
```

```
['a', 'b', 'c']
['1', '2', '3']
['1', '2', '3', '4']
```

From there, it's up to you to do the wrangling necessary to put the data in the form that you need it. For example:

In [45]:

```
lines = list(csv.reader(open('ex7.csv')))
header, values = lines[0], lines[1:]
data_dict = {h: v for h, v in zip(header, zip(*values))}
data_dict
```

Out[45]:

```
{'a': ('1', '1'), 'b': ('2', '2'), 'c': ('3', '3')}
```

CSV files come in many different flavors. Defining a new format with a different de- limiter, string quoting convention, or line terminator is done by defining a simple sub- class of csv.Dialect :

In [51]:

```
f = open("mydata.csv")
reader = csv.reader(f)
class my_dialect(csv.Dialect):
    lineterminator = '\n'
    delimiter = ';'
    quotechar = '"'
    quoting = csv.QUOTE_MINIMAL
reader = csv.reader(f, dialect=my_dialect)
```

The possible options (attributes of csv.Dialect) and what they do can be found in Table 6-3.

Table 6-3. CSV dialect options

Argument	Description
`delimiter`	One-character string to separate fields. Defaults to `','`.
`lineterminator`	Line terminator for writing, defaults to `'\r\n'`. Reader ignores this and recognizes cross-platform line terminators.
`quotechar`	Quote character for fields with special characters (like a delimiter). Default is `'"'`.
`quoting`	Quoting convention. Options include `csv.QUOTE_ALL` (quote all fields), `csv.QUOTE_MINIMAL` (only fields with special characters like the delimiter),

Argument	Description
	`csv.QUOTE_NONNUMERIC`, and `csv.QUOTE_NON` (no quoting). See Python's documentation for full details. Defaults to `QUOTE_MINIMAL`.
`skipinitialspace`	Ignore whitespace after each delimiter. Default `False`.
`doublequote`	How to handle quoting character inside a field. If `True`, it is doubled. See online documentation for full detail and behavior.
`escapechar`	String to escape the delimiter if `quoting` is set to `csv.QUOTE_NONE`. Disabled by default

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 [52]:

```
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'))
```

JSON Data

JSON (short for JavaScript Object Notation) has become one of the standard formats for sending data by HTTP request between web browsers and other applications. It is a much more flexible data format than a tabular text form like CSV. Here is an example:

In [53]:

```
obj = """
{"name": "Wes",
"places_lived": ["United States", "Spain", "Germany"],
"pet": null,
"siblings": [{"name": "Scott", "age": 25, "pet": "Zuko"},
{"name": "Katie", "age": 33, "pet": "Cisco"}]
}
"""
```

JSON is very nearly valid Python code with the exception of its null value null and some other nuances (such as disallowing trailing commas at the end of lists). The basic types are objects (dicts), arrays (lists), strings, numbers, booleans, and nulls. All of the keys in an object must be strings. There are several Python libraries for reading and writing JSON data. I'll use json here as it is built into the Python standard library. To convert a JSON string to Python form, use json.loads :

In [55]:

```
import json
result = json.loads(obj)
result
```

Out[55]:

```
{'name': 'Wes',
 'places_lived': ['United States', 'Spain', 'Germany'],
 'pet': None,
 'siblings': [{'name': 'Scott', 'age': 25, 'pet': 'Zuko'},
  {'name': 'Katie', 'age': 33, 'pet': 'Cisco'}]}
```

json.dumps on the other hand converts a Python object back to JSON:

In [56]:

```
asjson = json.dumps(result)
```

How you convert a JSON object or list of objects to a DataFrame or some other data structure for analysis will be up to you. Conveniently, you can pass a list of JSON objects to the DataFrame constructor and select a subset of the data fields:

In [58]:

```
siblings = pd.DataFrame(result['siblings'], columns=['name', 'age'])
siblings
```

Out[58]:

	name	age
0	Scott	25
1	Katie	33

Binary Data Formats

One of the easiest ways to store data efficiently in binary format is using Python's built- in pickle serialization. Conveniently, pandas objects all have a save method which writes the data to disk as a pickle:

In [2]:

```
import pandas as pd
frame = pd.read_csv('ex1.csv')
frame
```

Out[2]:

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

In [3]:

```
frame.to_pickle('frame_pickle')
```

Additional https://www.datacamp.com/community/tutorials/reading-writing-files-python