CSCI 303

Introduction to Data Science

11 - Data Acquisition

This Lecture

- Importing data using Python and pandas

The obligatory setup code...

In [2]:

```
import numpy as np
import pandas as pd

from pandas import Series, DataFrame
```

Raw Python

Text-based files are easy to read and write in Python.

In particular, files which data is organized into individual lines.

There's a sample file, `text.txt` , located in the directory `11-acquisition-files` .

We can view it by asking Jupyter to execute the linux command `cat` on the file (only works if you are not using Windows):

In [3]:

```
!cat 11-acquisition-files/text.txt
```

```
Name    Age     Salary  Hired
Laura   52      103,790 1/1/2005
Shashi  46      89,100  6/16/2010
Jun     33      85,500  7/1/2017
Bruce   48      96,445  12/1/2008
Raluca  40      110,080 9/15/2012
```

Reading the file via Python is pretty simple (use this if you are using a Windows operating system):

In [4]:

```
f = open('11-acquisition-files/text.txt') # get an open file object
for line in f:                            # for loops work!
    print(line, end='')                   # print without an endline

f.close()
```

```
Name    Age     Salary  Hired
Laura   52      103,790 1/1/2005
Shashi  46      89,100  6/16/2010
Jun     33      85,500  7/1/2017
Bruce   48      96,445  12/1/2008
Raluca  40      110,080 9/15/2012
```

Python has many tools to let us relatively easily parse files like this.

For simple files, the string `split` method may suffice:

In [5]:

```
f = open('11-acquisition-files/text.txt')
for line in f:
    print(line.split())
f.close()
```

```
['Name', 'Age', 'Salary', 'Hired']
['Laura', '52', '103,790', '1/1/2005']
['Shashi', '46', '89,100', '6/16/2010']
['Jun', '33', '85,500', '7/1/2017']
['Bruce', '48', '96,445', '12/1/2008']
['Raluca', '40', '110,080', '9/15/2012']
```

If you know regular expressions, you can parse more complicated files.

However, you're unlikely to need something that complicated.

Let's bring this data into pandas, the hard way:

In [6]:

```
data = []
f = open('11-acquisition-files/text.txt')
columns = next(f).split()
for line in f:
    data.append(line.split())
f.close()
df = DataFrame(data=data, columns=columns)
df
```

Out[6]:

	Name	Age	Salary	Hired
0	Laura	52	103,790	1/1/2005
1	Shashi	46	89,100	6/16/2010
2	Jun	33	85,500	7/1/2017
3	Bruce	48	96,445	12/1/2008
4	Raluca	40	110,080	9/15/2012

pandas `read_table`

For many file formats, it is easiest to let pandas do all the work!

One of the most basic tools is used to read text files like the one we worked with above is `read_table` .

In [7]:

```
pd.read_table?
```

We don't need to do much for this file, as it is nicely tab-delimited and has a header row.

In [8]:

```
df2 = pd.read_table('11-acquisition-files/text.txt')
df2
```

Out[8]:

	Name	Age	Salary	Hired
0	Laura	52	103,790	1/1/2005
1	Shashi	46	89,100	6/16/2010
2	Jun	33	85,500	7/1/2017
3	Bruce	48	96,445	12/1/2008
4	Raluca	40	110,080	9/15/2012

Our table might not have a header row, in which case we have to supply column labels:

In [9]:

```
df3 = pd.read_table('11-acquisition-files/text-no-header.txt',
                    header=None, names=['Person','Age','Income'])
df3
```

Out[9]:

	Person	Age	Income
0	Laura	52	103,790
1	Shashi	46	89,100
2	Jun	33	85,500
3	Bruce	48	96,445
4	Raluca	40	110,080

There are lots of other ways to manipulate the data when reading.

For instance, we can make one column the row index:

In [10]:

```python
df4 = pd.read_table('11-acquisition-files/text.txt',
                    index_col=0)
df4
```

Out[10]:

	Age	Salary	Hired
Name			
Laura	52	103,790	1/1/2005
Shashi	46	89,100	6/16/2010
Jun	33	85,500	7/1/2017
Bruce	48	96,445	12/1/2008
Raluca	40	110,080	9/15/2012

We can also do this by giving the column header:

In [11]:

```python
df5 = pd.read_table('11-acquisition-files/text.txt',
                    index_col='Name')
df5
```

Out[11]:

	Age	Salary	Hired
Name			
Laura	52	103,790	1/1/2005
Shashi	46	89,100	6/16/2010
Jun	33	85,500	7/1/2017
Bruce	48	96,445	12/1/2008
Raluca	40	110,080	9/15/2012

If tabs are not your separator, you can specify separators either as a specific character, or using a regular expression.

For arbitrary length whitespace separation, use `'\s+'` :

In [12]:

```python
df6 = pd.read_table('11-acquisition-files/text.txt', sep='\s+')
df6
```

Out[12]:

	Name	Age	Salary	Hired
0	Laura	52	103,790	1/1/2005
1	Shashi	46	89,100	6/16/2010
2	Jun	33	85,500	7/1/2017
3	Bruce	48	96,445	12/1/2008
4	Raluca	40	110,080	9/15/2012

Don't want the whole thing?

Check out the `nrows` and `skiprows` arguments:

In [13]:

```
df7 = pd.read_table('11-acquisition-files/text.txt', nrows=3)
df7
```

Out[13]:

	Name	Age	Salary	Hired
0	Laura	52	103,790	1/1/2005
1	Shashi	46	89,100	6/16/2010
2	Jun	33	85,500	7/1/2017

In [14]:

```
df8 = pd.read_table('11-acquisition-files/text.txt',
                    skiprows=[0,1])
df8
```

Out[14]:

	Shashi	46	89,100	6/16/2010
0	Jun	33	85,500	7/1/2017
1	Bruce	48	96,445	12/1/2008
2	Raluca	40	110,080	9/15/2012

Type Inference in `read_table`

Let's take a closer look at the DataFrame objects we're getting back:

In [15]:

```
df9 = pd.read_table('11-acquisition-files/text.txt')
print(df9.info())
df9
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
Name      5 non-null object
Age       5 non-null int64
Salary    5 non-null object
Hired     5 non-null object
dtypes: int64(1), object(3)
memory usage: 240.0+ bytes
None
```

Out[15]:

	Name	Age	Salary	Hired
0	Laura	52	103,790	1/1/2005
1	Shashi	46	89,100	6/16/2010
2	Jun	33	85,500	7/1/2017
3	Bruce	48	96,445	12/1/2008
4	Raluca	40	110,080	9/15/2012

It correctly *inferred* the type of the Age column as an integer.

But it didn't pick up Salary as a number, which we kind of want.

OK, this can be fixed!

In [16]:

```
df10 = pd.read_table('11-acquisition-files/text.txt',
                     thousands=',')
df10.info()
#pd.read_table?
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
Name      5 non-null object
Age       5 non-null int64
Salary    5 non-null int64
Hired     5 non-null object
dtypes: int64(2), object(2)
memory usage: 240.0+ bytes
```

Dates are also something pandas knows about:

In [17]:

```
df11 = pd.read_table('11-acquisition-files/text.txt',
                     thousands=',',
                     parse_dates=['Hired'])
df11.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 4 columns):
Name      5 non-null object
Age       5 non-null int64
Salary    5 non-null int64
Hired     5 non-null datetime64[ns]
dtypes: datetime64[ns](1), int64(2), object(1)
memory usage: 240.0+ bytes
```

The display data now looks slightly different:

In [18]:

```
df11
```

Out[18]:

	Name	Age	Salary	Hired
0	Laura	52	103790	2005-01-01
1	Shashi	46	89100	2010-06-16
2	Jun	33	85500	2017-07-01
3	Bruce	48	96445	2008-12-01
4	Raluca	40	110080	2012-09-15

pandas `read_csv`

A startling amount of the time, data is available in *comma-separated values* format.

The CSV format is commonly used to exchange data between things like spreadsheets and databases.

Here's what our data might look like exported from a spreadsheet program:

In [19]:

```
!cat 11-acquisition-files/text.csv
```

```
Name,Age,Salary,Hired
Laura,52,103790,1/1/2005
Shashi,46,89100,6/16/2010
Jun,33,85500,7/1/2017
Bruce,48,96445,12/1/2008
Raluca,40,110080,9/15/2012
```

pandas `read_csv` is pretty much identical to `read_table`, but assumes a CSV format:

In [20]:

```
df12 = pd.read_csv('11-acquisition-files/text.csv',
                   parse_dates=['Hired'])
df12
```

Out[20]:

	Name	Age	Salary	Hired
0	Laura	52	103790	2005-01-01
1	Shashi	46	89100	2010-06-16
2	Jun	33	85500	2017-07-01
3	Bruce	48	96445	2008-12-01
4	Raluca	40	110080	2012-09-15

JSON Data

Increasingly data is available in JSON (*JavaScript Object Notation*) format.

JSON is a format for exchanging rich, structured data as plain text.

JSON object strings look (and act) remarkably like Python code for dictionaries:

{ "Name" : "Laura", "Age" : 52, "Salary" : 103790, "Pets" : [{ "type" : "rabbit", "name" : "Gandalf" }, { "type" : "dog", "name" : "Aragorn" }] }

JSON basic types include strings, numbers (integer or floating point), Booleans, and nulls.

Compound types are objects and lists, which correspond pretty directly to Python dicts and lists.

Python has a library for interpreting JSON strings and turning them into Python objects (and vice versa):

In [21]:

```
import json

s = '''
   { "Name" : "Laura",
     "Age" : 52,
     "Salary" : 103790,
     "Pets" : [
       { "type" : "rabbit", "name" : "Gandalf" },
       { "type" : "dog", "name" : "Aragorn" } ] }
    '''

obj = json.loads(s)
obj
```

Out[21]:

```
{'Name': 'Laura',
 'Age': 52,
 'Salary': 103790,
 'Pets': [{'type': 'rabbit', 'name': 'Gandalf'},
  {'type': 'dog', 'name': 'Aragorn'}]}
```

It turns out that pandas DataFrame objects can be created from Python dictionaries, although its approach isn't always exactly what you expect:

In [22]:

```
df13 = DataFrame(obj)
df13
```

Out[22]:

	Name	Age	Salary	Pets
0	Laura	52	103790	{'type': 'rabbit', 'name': 'Gandalf'}
1	Laura	52	103790	{'type': 'dog', 'name': 'Aragorn'}

When dictionaries are fairly "flat", pandas interprets them pretty sensibly.

The two structures it handles best are dictionaries where the keys represent columns, and lists, where the entries are dictionaries representing rows.

Here's an example for the first:

In [23]:

```
d =   {"Name"   : ["Laura","Shashi","Jun","Bruce","Raluca"],
       "Age"    : [52,46,33,48,40],
       "Salary" : [103790,89100,85500,96445,110080],
       "Hired"  : ["1/1/2005","6/16/2010","7/1/2017","12/1/2008","9/15/2012"]}
DataFrame(d)
```

Out[23]:

	Name	Age	Salary	Hired
0	Laura	52	103790	1/1/2005
1	Shashi	46	89100	6/16/2010
2	Jun	33	85500	7/1/2017
3	Bruce	48	96445	12/1/2008
4	Raluca	40	110080	9/15/2012

and the second:

In [24]:

```python
d2 = [
    {"Name" : "Laura", "Age" : 52, "Salary" : 103790, "Hired" : "1/1/2005"},
    {"Name" : "Shashi", "Age" : 46, "Salary" : 89100, "Hired" : "6/16/2010"},
    {"Name" : "Jun", "Age" : 33, "Salary" : 85500, "Hired" : "7/1/2017"},
    {"Name" : "Bruce", "Age" : 48, "Salary" : 96445, "Hired" : "12/1/2008"},
    {"Name" : "Raluca", "Age" : 40, "Salary" : 110080, "Hired" : "9/15/2012"}
]
DataFrame(d2)
```

Out[24]:

	Age	Hired	Name	Salary
0	52	1/1/2005	Laura	103790
1	46	6/16/2010	Shashi	89100
2	33	7/1/2017	Jun	85500
3	48	12/1/2008	Bruce	96445
4	40	9/15/2012	Raluca	110080

In both cases, pandas re-ordered the columns alphabetically; we can tell it what order we want things in:

In [25]:

```python
DataFrame(d2, columns=['Name','Age','Salary','Hired'])
```

Out[25]:

	Name	Age	Salary	Hired
0	Laura	52	103790	1/1/2005
1	Shashi	46	89100	6/16/2010
2	Jun	33	85500	7/1/2017
3	Bruce	48	96445	12/1/2008
4	Raluca	40	110080	9/15/2012

Additional structures are possible, such as ones in which row index labels are explicitly provided:

In [26]:

```python
d3 =  {"Age"     : {"Laura" : 52, "Shashi" : 46},
       "Salary"  : {"Laura" : 103790, "Shashi" : 89100},
       "Hired"   : {"Laura" : "1/1/2005", "Shashi" : "6/16/2010"}}
DataFrame(d3)
```

Out[26]:

	Age	Salary	Hired
Laura	52	103790	1/1/2005
Shashi	46	89100	6/16/2010

Not surprisingly, then, if data is stored in JSON in any of these structures, pandas can read it quite easily.

Here is our simple data set in JSON data formats:

In [27]:

```
!cat 11-acquisition-files/json1.json
d14 = pd.read_json('11-acquisition-files/json1.json')
d14
```

```
[{"Name" : "Laura", "Age" : 52, "Salary" : 103790, "Hired" : "1/1/2005"},
 {"Name" : "Shashi", "Age" : 46, "Salary" : 89100, "Hired" : "6/16/2010"},
 {"Name" : "Jun", "Age" : 33, "Salary" : 85500, "Hired" : "7/1/2017"},
 {"Name" : "Bruce", "Age" : 48, "Salary" : 96445, "Hired" : "12/1/2008"},
 {"Name" : "Raluca", "Age" : 40, "Salary" : 110080, "Hired" : "9/15/2012"}]
```

Out[27]:

	Age	Hired	Name	Salary
0	52	1/1/2005	Laura	103790
1	46	6/16/2010	Shashi	89100
2	33	7/1/2017	Jun	85500
3	48	12/1/2008	Bruce	96445
4	40	9/15/2012	Raluca	110080

In [28]:

```
!cat 11-acquisition-files/json2.json
d15 = pd.read_json('11-acquisition-files/json2.json',
                   convert_dates=['Hired'])
d15
```

```
{"Name"   : ["Laura","Shashi","Jun","Bruce","Raluca"],
 "Age"    : [52,46,33,48,40],
 "Salary" : [103790,89100,85500,96445,110080],
 "Hired"  : ["1/1/2005","6/16/2010","7/1/2017","12/1/2008","9/15/2012"]}
```

Out[28]:

	Name	Age	Salary	Hired
0	Laura	52	103790	2005-01-01
1	Shashi	46	89100	2010-06-16
2	Jun	33	85500	2017-07-01
3	Bruce	48	96445	2008-12-01
4	Raluca	40	110080	2012-09-15

Oddly, there doesn't seem to be a way to set the order of columns in `read_json`, but we can reorder pretty easily:

In [29]:

```
d16 = d15[['Name','Age','Salary','Hired']]
d16
```

Out[29]:

	Name	Age	Salary	Hired
0	Laura	52	103790	2005-01-01
1	Shashi	46	89100	2010-06-16
2	Jun	33	85500	2017-07-01
3	Bruce	48	96445	2008-12-01
4	Raluca	40	110080	2012-09-15