Intro to pandas data structures

October 26, 2013 | Tags: python pandas sql tutorial data science

UPDATE: If you're interested in learning pandas from a SQL perspective and would prefer to watch a video, you can find video of my 2014 PyData NYC talk here.

However, the other week a couple of coworkers expressed their interest in learning a bit more about it - this seemed like a good reason to revisit the topic.

What follows is a fairly thorough introduction to the library. I chose to break it into three parts as I felt it was too long and daunting as one.

- Part 1: Intro to pandas data structures, covers the basics of the library's two main data structures - Series and DataFrames.
- Part 2: Working with DataFrames, dives a bit deeper into the functionality of DataFrames. It shows how to inspect, select, filter, merge, combine, and group your data.
- Part 3: Using pandas with the MovieLens dataset, applies the learnings of the first two parts in order to answer a few basic analysis questions about the MovieLens ratings data.

If you'd like to follow along, you can find the necessary CSV files here and the MovieLens dataset here.

library for data analysis. Python has always been great for prepping and munging data, but it's never been great for analysis - you'd usually end up using R or loading it into a database and using SQL (or worse, Excel). pandas makes Python great for analysis.

Data Structures

pandas introduces two new data structures to Python - Series and DataFrame, both of which are built on top of NumPy (this means it's fast).

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
pd.set_option('max_columns', 50)
%matplotlib inline
```

Series

A Series is a one-dimensional object similar to an array, list, or column in a table. It will assign a labeled index to each item in the Series. By default, each item will receive an index label from 0 to N, where N is the length of the Series minus one.

```
0                7
1       Heisenberg
2             3.14
3      -1789710578
4    Happy Eating!
dtype: object
```

Alternatively, you can specify an index to use when creating the Series.

```
s = pd.Series([7, 'Heisenberg', 3.14, -178971057
              index=['A', 'Z', 'C', 'Y', 'E'])
s
```

```
A                7
Z       Heisenberg
C             3.14
Y      -1789710578
E    Happy Eating!
dtype: object
```

The Series constructor can convert a dictonary as well, using the keys of the dictionary as its index.

```
d = {'Chicago': 1000, 'New York': 1300, 'Portlan
     'Austin': 450, 'Boston': None}
cities = pd.Series(d)
cities
```

```
Austin            450
Boston            NaN
Chicago          1000
New York         1300
Portland          900
San Francisco    1100
dtype: float64
```

You can use the index to select specific items from the Series ...

```
cities['Chicago']
```

```
1000.0
```

```
Chicago          1000
Portland          900
San Francisco    1100
dtype: float64
```

Or you can use boolean indexing for selection.

```
cities[cities < 1000]
```

```
Austin      450
Portland    900
dtype: float64
```

That last one might be a little weird, so let's make it more clear - cities < 1000 returns a Series of True/False values, which we then pass to our Series cities, returning the corresponding True items.

```
less_than_1000 = cities < 1000
print(less_than_1000)
print('\n')
print(cities[less_than_1000])
```

```
Austin            True
Boston           False
Chicago          False
New York         False
Portland          True
San Francisco    False
dtype: bool


Austin      450
Portland    900
dtype: float64
```

You can also change the values in a Series on the fly.

```
('Old value:', 1000.0)
('New value:', 1400.0)
```

```
# changing values using boolean logic
print(cities[cities < 1000])
print('\n')
cities[cities < 1000] = 750

print cities[cities < 1000]
```

```
Austin      450
Portland    900
dtype: float64


Austin      750
Portland    750
dtype: float64
```

What if you aren't sure whether an item is in the Series? You can check using idiomatic Python.

```
print('Seattle' in cities)
print('San Francisco' in cities)
```

```
False
True
```

Mathematical operations can be done using scalars and functions.

```
# divide city values by 3
cities / 3
```

```
Austin           250.000000
Boston                  NaN
Chicago          466.666667
New York         433.333333
Portland         250.000000
San Francisco    366.666667
dtype: float64
```

```
Austin            562500
Boston               NaN
Chicago          1960000
New York         1690000
Portland          562500
San Francisco    1210000
dtype: float64
```

You can add two Series together, which returns a union of the two Series with the addition occurring on the shared index values. Values on either Series that did not have a shared index will produce a NULL/NaN (not a number).

```
print(cities[['Chicago', 'New York', 'Portland']]
print('\n')
print(cities[['Austin', 'New York']])
print('\n')
print(cities[['Chicago', 'New York', 'Portland']]
```

```
Chicago     1400
New York    1300
Portland     750
dtype: float64


Austin       750
New York    1300
dtype: float64


Austin       NaN
Chicago      NaN
New York    2600
Portland     NaN
dtype: float64
```

NULL/NaN values.

NULL checking can be performed with isnull and notnull.

```
# returns a boolean series indicating which valu
cities.notnull()
```

```
Austin            True
Boston           False
Chicago           True
New York          True
Portland          True
San Francisco     True
dtype: bool
```

```
# use boolean logic to grab the NULL cities
print(cities.isnull())
print('\n')
print(cities[cities.isnull()])
```

```
Austin           False
Boston            True
Chicago          False
New York         False
Portland         False
San Francisco    False
dtype: bool


Boston   NaN
dtype: float64
```

structure comprised of rows and columns, akin to a spreadsheet, database table, or R's data.frame object. You can also think of a DataFrame as a group of Series objects that share an index (the column names).

For the rest of the tutorial, we'll be primarily working with DataFrames.

Reading Data

To create a DataFrame out of common Python data structures, we can pass a dictionary of lists to the DataFrame constructor.

Using the columns parameter allows us to tell the constructor how we'd like the columns ordered. By default, the DataFrame constructor will order the columns alphabetically (though this isn't the case when reading from a file - more on that next).

	year	team	wins	losses
0	2010	Bears	11	5
1	2011	Bears	8	8
2	2012	Bears	10	6
3	2011	Packers	15	1
4	2012	Packers	11	5
5	2010	Lions	6	10
6	2011	Lions	10	6
7	2012	Lions	4	12

Much more often, you'll have a dataset you want to read into a DataFrame. Let's go through several common ways of doing so.

CSV

Reading a CSV is as simple as calling the *read_csv* function. By default, the *read_csv* function expects the column separator to be a comma, but you can change that using the sep parameter.

```
%cd ~/Dropbox/tutorials/pandas/
```

```
/Users/gjreda/Dropbox (Personal)/tu
```

Greg Reda

BLOG

```
Year,Age,Tm,Lg,W,L,W-L%,ERA,G,GS,GI
1995,25,NYY,AL,5,3,.625,5.51,19,10,
1996,26,NYY,AL,8,3,.727,2.09,61,0,1
1997,27,NYY,AL,6,4,.600,1.88,66,0,5
1998,28,NYY,AL,3,0,1.000,1.91,54,0,
```

```
from_csv = pd.read_csv('mariano-rivera.csv')
from_csv.head()
```

	Year	Age	Tm	Lg	W	L	W
0	1995	25	NYY	AL	5	3	0.(
1	1996	26	NYY	AL	8	3	0.7
2	1997	27	NYY	AL	6	4	0.(
3	1998	28	NYY	AL	3	0	1.(
4	1999	29	NYY	AL	4	3	0.5

Our file had headers, which the function inferred upon reading in the file. Had we wanted to be more explicit, we could have passed header=None to the function along with a list of column names to use:

```
# Source: pro-football-reference.com/players/M/M
!head -n 5 peyton-passing-TDs-2012.csv
```

```
1,1,2012-09-09,DEN,,PIT,W 31-19,3,7
2,1,2012-09-09,DEN,,PIT,W 31-19,4,1
3,2,2012-09-17,DEN,@,ATL,L 21-27,2,
4,3,2012-09-23,DEN,,HOU,L 25-31,4,3
5,3,2012-09-23,DEN,,HOU,L 25-31,4,6
```

	num	game	date	team
0	1	1	2012-09-09	DEN
1	2	1	2012-09-09	DEN
2	3	2	2012-09-17	DEN
3	4	3	2012-09-23	DEN
4	5	3	2012-09-23	DEN

pandas' various *reader* functions have many parameters allowing you to do things like skipping lines of the file, parsing dates, or specifying how to handle NA/NULL datapoints.

There's also a set of *writer* functions for writing to a variety of formats (CSVs, HTML tables, JSON). They function exactly as you'd expect and are typically called to_format:

```
my_dataframe.to_csv('path_to_file.csv')
```

Take a look at the IO documentation to familiarize yourself with file reading/writing functionality.

do, too. Thankfully, pandas allows you to read and write Excel files, so you can easily read from Excel, write your code in Python, and then write back out to Excel - no need for VBA.

Reading Excel files requires the xlrd library. You can install it via pip (*pip install xlrd*).

Let's first write a DataFrame to Excel.

```
# this is the DataFrame we created from a dictio
football.head()
```

	year	team	wins	losses
0	2010	Bears	11	5
1	2011	Bears	8	8
2	2012	Bears	10	6
3	2011	Packers	15	1
4	2012	Packers	11	5

```
# since our index on the football DataFrame is m
football.to_excel('football.xlsx', index=False)
```

```
!ls -l *.xlsx
```

```
-rw-r--r--@ 1 gjreda  staff  5665 M
```

```
# delete the DataFrame
del football
```

	year	team	wins	losses
0	2010	Bears	11	5
1	2011	Bears	8	8
2	2012	Bears	10	6
3	2011	Packers	15	1
4	2012	Packers	11	5
5	2010	Lions	6	10
6	2011	Lions	10	6
7	2012	Lions	4	12

Database

pandas also has some support for reading/writing DataFrames directly from/to a database [docs]. You'll typically just need to pass a connection object or sqlalchemy engine to the read_sql or to_sql functions within the pandas.io module.

Note that to_sql executes as a series of INSERT INTO statements and thus trades speed for simplicity. If you're writing a large DataFrame to a database, it might be quicker to write the DataFrame to CSV and load that directly using the database's file import arguments.

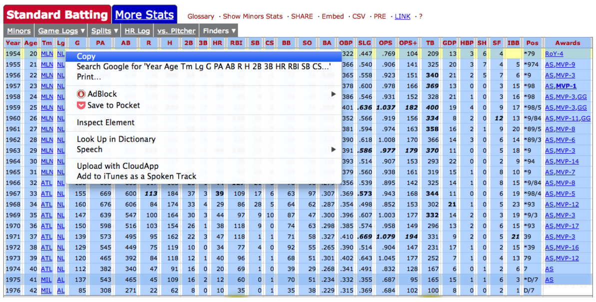
```
results = sql.read_sql(query, con=conn)
results.head()
```

	tow_date	make	style	mo
0	01/19/2013	FORD	LL	
1	01/19/2013	FORD	4D	
2	01/19/2013	FORD	4D	
3	01/19/2013	FORD	LL	
4	01/19/2013	FORD	LL	

read directly into a DataFrame, I prefer to read the results directly from the clipboard. I'm often tweaking queries in my SQL client (Sequel Pro), so I would rather see the results *before* I read it into pandas. Once I'm confident I have the data I want, then I'll read it into a DataFrame.

This works just as well with any type of delimited data you've copied to your clipboard. The function does a good job of inferring the delimiter, but you can also use the sep parameter to be explicit.

Hank Aaron



Greg Reda

BLOG

	Year	Age	Tm	Lg	G	PA
0	1954	20	MLN	NL	122	50
1	1955 ★	21	MLN	NL	153	66
2	1956 ★	22	MLN	NL	153	66
3	1957 ★	23	MLN	NL	151	67
4	1958 ★	24	MLN	NL	153	66

URL

With read_table, we can also read directly from a URL.

Let's use the best sandwiches data that I wrote about scraping a while back.

	rank	sandwich	restaurant
0	1	BLT	Old Oak Tap
1	2	Fried Bologna	Au Cheval
2	3	Woodland Mushroom	Xoco

Google Analytics

pandas also has some integration with the Google Analytics API, though there is some setup required. I won't be covering it, but you can read more about it here and here.

Move onto the next section, which covers working with DataFrames.

← **Back to Home**

Greg Reda

BLOG