Working with DataFrames

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.

This is part two of a three part introduction to pandas, a Python library for data analysis. The tutorial is primarily geared towards SQL users, but is useful for anyone wanting to get started with the library.

Part 1: Intro to pandas data structures

Part 2: Working with DataFrames

Part 3: Using pandas with the MovieLens dataset

Working with DataFrames

Greg Reda

BLOG

abundance of functionality, far too much for me to cover in this introduction. I'd encourage anyone interested in diving deeper into the library to check out its excellent documentation. Or just use Google - there are a lot of Stack Overflow questions and blog posts covering specifics of the library.

We'll be using the MovieLens dataset in many examples going forward. The dataset contains 100,000 ratings made by 943 users on 1,682 movies.

```
# pass in column names for each CSV
u_cols = ['user_id', 'age', 'sex', 'occupation',
users = pd.read_csv('ml-100k/u.user', sep='|', n
                    encoding='latin-1')

r_cols = ['user_id', 'movie_id', 'rating', 'unix
ratings = pd.read_csv('ml-100k/u.data', sep='\t'
                      encoding='latin-1')

# the movies file contains columns indicating th
# let's only load the first five columns of the
m_cols = ['movie_id', 'title', 'release_date', '
movies = pd.read_csv('ml-100k/u.item', sep='|',
                     encoding='latin-1')
```

Inspection

pandas has a variety of functions for getting basic information about your DataFrame, the most basic of which is using the info method.

```
<class 'pandas.core.frame.DataFrame
Int64Index: 1682 entries, 0 to 168
Data columns (total 5 columns):
movie_id              1682 non-null
title                 1682 non-null
release_date          1681 non-null
video_release_date    0 non-null fl
imdb_url              1679 non-null
dtypes: float64(1), int64(1), objec
memory usage: 78.8+ KB
```

The output tells a few things about our DataFrame.

1. It's obviously an instance of a DataFrame.
2. Each row was assigned an index of 0 to N-1, where N is the number of rows in the DataFrame. pandas will do this by default if an index is not specified. Don't worry, this can be changed later.
3. There are 1,682 rows (every row must have an index).
4. Our dataset has five total columns, one of which isn't populated at all (video_release_date) and two that are missing some values (release_date and imdb_url).
5. The last datatypes of each column, but not necessarily in the corresponding order to the listed columns. You should use the dtypes method to get the datatype for each column.
6. An approximate amount of RAM used to hold the DataFrame. See the .memory_usage method

```
movies.dtypes
```

```
movie_id                int64
title                  object
release_date           object
video_release_date    float64
imdb_url               object
dtype: object
```

numeric columns. Be careful though, since this will return information on all columns of a numeric datatype.

```
users.describe()
```

	user_id	age
count	943.000000	943.000000
mean	472.000000	34.051962
std	272.364951	12.192740
min	1.000000	7.000000
25%	236.500000	25.000000
50%	472.000000	31.000000
75%	707.500000	43.000000
max	943.000000	73.000000

Notice *user_id* was included since it's numeric. Since this is an ID value, the stats for it don't really matter.

We can quickly see the average age of our users is just above 34 years old, with the youngest being 7 and the oldest being 73. The median age is 31, with the youngest quartile of users being 25 or younger, and the oldest quartile being at least 43.

head displays the first five records of the dataset, while tail displays the last five.

```
movies.head()
```

	movie_id	title	release
0	1	Toy Story (1995)	01-Jan-
1	2	GoldenEye (1995)	01-Jan-
2	3	Four Rooms (1995)	01-Jan-
3	4	Get Shorty (1995)	01-Jan-
4	5	Copycat (1995)	01-Jan-

	movie_id	title	releas
1679	1680	Sliding Doors (1998)	01-Jan
1680	1681	You So Crazy (1994)	01-Jan
1681	1682	Scream of Stone (Schrei aus Stein) (1991)	08-Mar

Alternatively, Python's regular slicing syntax works as well.

```
movies[20:22]
```

	movie_id	title	releas
20	21	Muppet Treasure Island (1996)	16-Fel
21	22	Braveheart (1995)	16-Fel

group of Series that share an index (in this case the column headers). This makes it easy to select specific columns.

Selecting a single column from the DataFrame will return a Series object.

```
users['occupation'].head()
```

```
0    technician
1         other
2        writer
3    technician
4         other
Name: occupation, dtype: object
```

To select multiple columns, simply pass a list of column names to the DataFrame, the output of which will be a DataFrame.

```
print(users[['age', 'zip_code']].head())
print('\n')

# can also store in a variable to use later
columns_you_want = ['occupation', 'sex']
print(users[columns_you_want].head())
```

```
   age zip_code
0   24    85711
1   53    94043
2   23    32067
3   24    43537
4   33    15213


   occupation sex
0  technician   M
1       other   F
2      writer   M
3  technician   M
4       other   F
```

Greg Reda

BLOG

typically easiest.

```
# users older than 25
print(users[users.age > 25].head(3))
print('\n')

# users aged 40 AND male
print(users[(users.age == 40) & (users.sex == 'M
print('\n')

# users younger than 30 OR female
print(users[(users.sex == 'F') | (users.age < 30
```

```
   user_id  age sex occupation zip_
1        2   53   F      other    9
4        5   33   F      other    1
5        6   42   M  executive    9


     user_id  age sex  occupation z
18        19   40   M   librarian
82        83   40   M       other
115      116   40   M  healthcare


   user_id  age sex  occupation zip
0        1   24   M  technician
1        2   53   F       other
2        3   23   M      writer
```

Since our index is kind of meaningless right now, let's set it to the _user*id* using the set_index method. By default, set_index returns a new DataFrame, so you'll have to specify if you'd like the changes to occur in place.

This has confused me in the past, so look carefully at the code and output below.

```
with_new_index = users.set_index('user_id')
print(with_new_index.head())
print("\n^^^ set_index actually returns a new Da
```

```
         age sex  occupation zip_co
user_id
1         24   M  technician    857
2         53   F       other    940
3         23   M      writer    320
4         24   M  technician    435
5         33   F       other    152


   user_id  age sex  occupation zip
0        1   24   M  technician
1        2   53   F       other
2        3   23   M      writer
3        4   24   M  technician
4        5   33   F       other

^^^ I didn't actually change the Da

         age sex  occupation zip_co
user_id
1         24   M  technician    857
2         53   F       other    940
3         23   M      writer    320
4         24   M  technician    435
5         33   F       other    152

^^^ set_index actually returns a ne
```

If you want to modify your existing DataFrame, use the inplace parameter. Most DataFrame methods return new a DataFrames, while offering an inplace parameter. Note that the inplace version might not actually be any more efficint (in terms of speed or memory usage) that the regular version.

	age	sex	occupation
user_id			
1	24	M	technician
2	53	F	other
3	23	M	writer
4	24	M	technician
5	33	F	other

Notice that we've lost the default pandas 0-based index and moved the user_id into its place. We can select rows *by position* using the iloc method.

```
print(users.iloc[99])
print('\n')
print(users.iloc[[1, 50, 300]])
```

```
age                   36
sex                    M
occupation     executive
zip_code           90254
Name: 100, dtype: object


         age sex occupation zip_co
user_id
2         53   F      other    940
51        28   M   educator    165
301       24   M    student    554
```

And we can select rows *by label* with the loc method.

```
age                   36
sex                    M
occupation     executive
zip_code           90254
Name: 100, dtype: object
```

```
         age sex occupation zip_coc
user_id
2         53   F      other    940‹
51        28   M   educator    165(
301       24   M    student    554:
```

If we realize later that we liked the old pandas default index, we can just reset_index. The same rules for inplace apply.

```
users.reset_index(inplace=True)
users.head()
```

	user_id	age	sex	occupatio
0	1	24	M	technician
1	2	53	F	other
2	3	23	M	writer
3	4	24	M	technician
4	5	33	F	other

indexing

- Use iloc for positional indexing

I've found that I can usually get by with boolean indexing, loc and iloc, but pandas has a whole host of other ways to do selection.

typically stored in a relational manner.

Our MovieLens data is a good example of this - a rating requires both a user and a movie, and the datasets are linked together by a key - in this case, the user_id and movie_id. It's possible for a user to be associated with zero or many ratings and movies. Likewise, a movie can be rated zero or many times, by a number of different users.

Like SQL's JOIN clause, pandas.merge allows two DataFrames to be joined on one or more keys. The function provides a series of parameters (on, left_on, right_on, left_index, right_index) allowing you to specify the columns or indexes on which to join.

By default, pandas.merge operates as an *inner join*, which can be changed using the how parameter.

From the function's docstring:

> *how : {'left', 'right', 'outer', 'inner'}, default 'inner'*
>
> - *left: use only keys from left frame (SQL: left*

```
print(right_frame)
```

```
   key left_value
0    0          a
1    1          b
2    2          c
3    3          d
4    4          e


   key right_value
0    2           f
1    3           g
2    4           h
3    5           i
4    6           j
```

inner join (default)

```
pd.merge(left_frame, right_frame, on='key', how=
```

	key	left_value	right_value
0	2	c	f
1	3	d	g
2	4	e	h

Greg Reda

BLOG

```
SELECT left_frame.key, left_frame.left_
FROM left_frame
INNER JOIN right_frame
    ON left_frame.key = right_frame.key
```

Had our *key* columns not been named the same, we could have used the *left_on* and *right_on* parameters to specify which fields to join from each frame.

```
pd.merge(left_frame, right_frame, left_on
```

Alternatively, if our keys were indexes, we could use the left_index or right_index parameters, which accept a True/False value. You can mix and match columns and indexes like so:

```
pd.merge(left_frame, right_frame, left_on
```

left outer join

Greg Reda

BLOG

	key	left_value	right_value
0	0	a	NaN
1	1	b	NaN
2	2	c	f
3	3	d	g
4	4	e	h

We keep everything from the left frame, pulling in the value from the right frame where the keys match up. The right_value is NULL where keys do not match (NaN).

SQL Equivalent:

```
SELECT left_frame.key, left_frame.left_valu
FROM left_frame
LEFT JOIN right_frame
    ON left_frame.key = right_frame.key;
```

right outer join

```
pd.merge(left_frame, right_frame, on='key', how=
```

	key	left_value	right_value
0	2	c	f
1	3	d	g
2	4	e	h
3	5	NaN	i
4	6	NaN	j

key did not find a match.

SQL Equivalent:

```
SELECT right_frame.key, left_frame.left_val
FROM left_frame
RIGHT JOIN right_frame
    ON left_frame.key = right_frame.key;
```

full outer join

```
pd.merge(left_frame, right_frame, on='key', how=
```

	key	left_value	right_value
0	0	a	NaN
1	1	b	NaN
2	2	c	f
3	3	d	g
4	4	e	h
5	5	NaN	i
6	6	NaN	j

Greg Reda

BLOG

Where there was not a match, the values corresponding to that key are NULL.

SQL Equivalent (though some databases don't allow FULL JOINs (e.g. MySQL)):

```
SELECT IFNULL(left_frame.key, right_frame.k
        , left_frame.left_value, right_fram
FROM left_frame
FULL OUTER JOIN right_frame
    ON left_frame.key = right_frame.key;
```

Combining

pandas also provides a way to combine DataFrames along an axis - pandas.concat. While the function is equivalent to SQL's UNION clause, there's a lot more that can be done with it.

pandas.concat takes a list of Series or DataFrames and returns a Series or DataFrame of the concatenated objects. Note that because the function takes list, you can combine many objects at once.

	key	left_value	right_value
0	0	a	NaN
1	1	b	NaN
2	2	c	NaN
3	3	d	NaN
4	4	e	NaN
0	2	NaN	f
1	3	NaN	g
2	4	NaN	h
3	5	NaN	i
4	6	NaN	j

By default, the function will vertically append the objects to one another, combining columns with the same name. We can see above that values not matching up will be NULL.

Additionally, objects can be concatentated side-by-side using the function's *axis* parameter.

```
pd.concat([left_frame, right_frame], axis=1)
```

	key	left_value	key	right_va
0	0	a	2	f
1	1	b	3	g
2	2	c	4	h
3	3	d	5	i
4	4	e	6	j

Series/DataFrames into one unified object. The documentation has some examples on the ways it can be used.

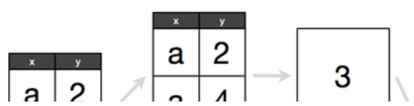
for me to grasp, but it's pretty awesome once it clicks.

pandas groupby method draws largely from the split-apply-combine strategy for data analysis. If you're not familiar with this methodology, I highly suggest you read up on it. It does a great job of illustrating how to properly think through a data problem, which I feel is more important than any technical skill a data analyst/scientist can possess.

When approaching a data analysis problem, you'll often break it apart into manageable pieces, perform some operations on each of the pieces, and then put everything back together again (this is the gist split-apply-combine strategy). pandas groupby is great for these problems (R users should check out the plyr and dplyr packages).

If you've ever used SQL's GROUP BY or an Excel Pivot Table, you've thought with this mindset, probably without realizing it.

Assume we have a DataFrame and want to get the average for each group - visually, the split-apply-combine method looks like this:



some basic groupby examples using this data.

```
!head -n 3 city-of-chicago-salaries.csv
```

```
Name,Position Title,Department,Empl
"AARON,  ELVIA J",WATER RATE TAKER,
"AARON,  JEFFERY M",POLICE OFFICER,
```

Since the data contains a dollar sign for each salary, python will treat the field as a series of strings. We can use the converters parameter to change this when reading in the file.

> *converters : dict. optional*
>
> - *Dict of functions for converting values in certain columns. Keys can either be integers or column labels*

	name	title
0	AARON, ELVIA J	WATER RATE TAKER
1	AARON, JEFFERY M	POLICE OFFICER
2	AARON, KIMBERLEI R	CHIEF CONTRACT EXPEDITER
3	ABAD JR, VICENTE M	CIVIL ENGINEER IV
4	ABBATACOLA, ROBERT J	ELECTRICAL MECHANIC

pandas groupby returns a DataFrameGroupBy object which has a variety of methods, many of which are similar to standard SQL aggregate functions.

```
by_dept = chicago.groupby('department')
by_dept
```

```
<pandas.core.groupby.DataFrameGrour
```

Calling count returns the total number of NOT NULL values within each column. If we were interested in the total number of records in each group, we could use size.

```
                     name  title  sal
department
ADMIN HEARNG           42     42
ANIMAL CONTRL          61     61
AVIATION             1218   1218    :
BOARD OF ELECTION     110    110
BOARD OF ETHICS         9      9
```

```
department
PUBLIC LIBRARY     926
STREETS & SAN     2070
TRANSPORTN        1168
TREASURER           25
WATER MGMNT       1857
dtype: int64
```

Summation can be done via sum, averaging by mean, etc. (if it's a SQL function, chances are it exists in pandas). Oh, and there's median too, something not available in most databases.

```
                         salary
department
HUMAN RESOURCES       4850928.0
INSPECTOR GEN         4035150.0
IPRA                  7006128.0
LAW                  31883920.2
LICENSE APPL COMM       65436.0


                           salary
department
HUMAN RESOURCES      71337.176471
INSPECTOR GEN        80703.000000
IPRA                 82425.035294
LAW                  70853.156000
LICENSE APPL COMM    65436.000000


                      salary
department
HUMAN RESOURCES        68496
INSPECTOR GEN          76116
IPRA                   82524
LAW                    66492
LICENSE APPL COMM      65436
```

Operations can also be done on an individual Series within a grouped object. Say we were curious about the five departments with the most distinct titles - the pandas equivalent to:

```
SELECT department, COUNT(DISTINCT title)
FROM chicago
GROUP BY department
ORDER BY 2 DESC
LIMIT 5;
```

pandas is a lot less verbose here ...

```
department
WATER MGMNT    153
TRANSPORTN     150
POLICE         130
AVIATION       125
HEALTH         118
Name: title, dtype: int64
```

What if we wanted to see the highest paid employee within each department. Given our current dataset, we'd have to do something like this in SQL:

```
SELECT *
FROM chicago c
INNER JOIN (
    SELECT department, max(salary) max_sal
    FROM chicago
    GROUP BY department
) m
ON c.department = m.department
AND c.salary = m.max_salary;
```

This would give you the highest paid person in each department, but it would return multiple if there were many equally high paid people within a department.

Alternatively, you could alter the table, add a column, and then write an update statement to populate that column. However, that's not always an option.

Note: This would be a lot easier in PostgreSQL, T-SQL, and possibly Oracle due to the existence of partition/window/analytic functions. I've chosen to use MySQL syntax throughout this tutorial because of it's popularity. Unfortunately, MySQL doesn't have similar functions.

where N is the number of employees within the department. We can then call apply to, well, *apply* that function to each group (in this case, each department).

```
def ranker(df):
    """Assigns a rank to each employee based on
    Assumes the data is DESC sorted."""
    df['dept_rank'] = np.arange(len(df)) + 1
    return df
```

```
chicago.sort_values('salary', ascending=False, i
chicago = chicago.groupby('department').apply(ra
print(chicago[chicago.dept_rank == 1].head(7))
```

```
                      name
18039     MC CARTHY,  GARRY F  SUPE
8004           EMANUEL,  RAHM
25588        SANTIAGO,  JOSE A
763     ANDOLINO,  ROSEMARIE S  COMM
4697      CHOUCAIR,  BECHARA N    CC
21971       PATTON,  STEPHEN R
12635       HOLT,  ALEXANDRA D

       salary  dept_rank
18039  260004          1
8004   216210          1
25588  202728          1
763    186576          1
4697   177156          1
21971  173664          1
12635  169992          1
```

Move onto part three, using pandas with the MovieLens dataset.

← **Back to Home**

Greg Reda

BLOG