1 Working with DataFrames - by Greg Reda

- Now that we can get data into a DataFrame, we can finally start working with them.
- 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', 'zip_code']
users = pd.read_csv('ml-100k/u.user', sep='|', names=u_cols)

r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
ratings = pd.read_csv('ml-100k/u.data', sep='\t', names=r_cols)

# the movies file contains columns indicating the movie's genres
# let's only load the first five columns of the file with usecols
m_cols = ['movie_id', 'title', 'release_date', 'video_release_date', 'imdb_url']
movies = pd.read_csv('ml-100k/u.item', sep='|', names=m_cols,
usecols=range(5))
```

1.1 Inspection of the Data Set

pandas has a variety of functions for getting basic information about your DataFrame, the most basic of which is calling your DataFrame by name.

```
movies
Out[32]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1682 entries, 0 to 1681
Data columns (total 5 columns):
movie_id              1682  non-null values
title                 1682  non-null values
release_date          1681  non-null values
video_release_date    0  non-null values
imdb_url              1679  non-null values
dtypes: float64(1), int64(1), object(3)
```

The output tells a few things about our DataFrame.

- It is eveidently an instance of a DataFrame.
- 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.

- There are 1,682 rows (every row must have an index).
- 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`).
- The last line displays the 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.

```
movies.dtypes
```

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

DataFrame's also have a describe method, which is great for seeing basic statistics about the dataset's 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.

You've probably noticed that I've used the head method regularly throughout this post - by default, head displays the first five records of the dataset, while tail displays the last five.

```
print movies.head()
   movie_id              title release_date  video_release_date  \
0         1   Toy Story (1995)  01-Jan-1995                 NaN
1         2   GoldenEye (1995)  01-Jan-1995                 NaN
2         3  Four Rooms (1995)  01-Jan-1995                 NaN
3         4  Get Shorty (1995)  01-Jan-1995                 NaN
4         5     Copycat (1995)  01-Jan-1995                 NaN

                                          imdb_url
0  http://us.imdb.com/M/title-exact?Toy%20Story%2...
1  http://us.imdb.com/M/title-exact?GoldenEye%20(...
2  http://us.imdb.com/M/title-exact?Four%20Rooms%...
3  http://us.imdb.com/M/title-exact?Get%20Shorty%...
4  http://us.imdb.com/M/title-exact?Copycat%20(1995)
```

```
print movies.tail(3)
      movie_id                                      title release_date  \
1679      1680                      Sliding Doors (1998)  01-Jan-1998
1680      1681                       You So Crazy (1994)  01-Jan-1994
1681      1682  Scream of Stone (Schrei aus Stein) (1991)  08-Mar-1996

      video_release_date                                         imdb_url
1679                NaN      http://us.imdb.com/Title?Sliding+Doors+(1998)
1680                NaN  http://us.imdb.com/M/title-exact?You%20So%20Cr...
1681                NaN  http://us.imdb.com/M/title-exact?Schrei%20aus%...
```

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

```
print movies[20:22]
    movie_id                        title release_date  video_release_date  \
20        21  Muppet Treasure Island (1996)  16-Feb-1996                 NaN
21        22              Braveheart (1995)  16-Feb-1996                 NaN

                                          imdb_url
20  http://us.imdb.com/M/title-exact?Muppet%20Trea...
21  http://us.imdb.com/M/title-exact?Braveheart%20...
```

1.2 Selecting

You can think of a DataFrame as a 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()
Out[38]:
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
```

Row selection can be done multiple ways, but doing so by an individual index or boolean indexing are 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')].head(3)
print '\n'

# users younger than 30 OR female
print users[(users.sex == 'F') | (users.age < 30)].head(3)
   user_id  age sex occupation zip_code
1        2   53   F      other    94043
4        5   33   F      other    15213
5        6   42   M  executive    98101


     user_id  age sex  occupation zip_code
18        19   40   M   librarian    02138
82        83   40   M       other    44133
115      116   40   M  healthcare    97232


   user_id  age sex  occupation zip_code
0        1   24   M  technician    85711
1        2   53   F       other    94043
2        3   23   M      writer    32067
```

Since our index is kind of meaningless right now, let's set it to the userid_ 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.

```
print users.set_index('user_id').head()
print '\n'

print users.head()
print "\n^^^ I didn't actually change the DataFrame. ^^^\n"

with_new_index = users.set_index('user_id')
print with_new_index.head()
```

```
print "\n^^^ set_index actually returns a new DataFrame. ^^^\n"
         age sex  occupation zip_code
user_id
1         24   M  technician    85711
2         53   F       other    94043
3         23   M      writer    32067
4         24   M  technician    43537
5         33   F       other    15213
```

```
   user_id  age sex  occupation zip_code
0        1   24   M  technician    85711
1        2   53   F       other    94043
2        3   23   M      writer    32067
3        4   24   M  technician    43537
4        5   33   F       other    15213

^^^ I didn't actually change the DataFrame. ^^^

         age sex  occupation zip_code
user_id
1         24   M  technician    85711
2         53   F       other    94043
3         23   M      writer    32067
4         24   M  technician    43537
5         33   F       other    15213

^^^ set_index actually returns a new DataFrame. ^^^
```

If you want to modify your existing DataFrame, use the inplace parameter.

```
users.set_index('user_id', inplace=True)
print users.head()
         age sex  occupation zip_code
user_id
1         24   M  technician    85711
2         53   F       other    94043
3         23   M      writer    32067
```

```
4         24   M  technician    43537
5         33   F       other    15213
```

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

```
print users.ix[99]
print '\n'
print users.ix[[1, 50, 300]]
age                20
sex                 M
occupation    student
zip_code        63129
Name: 99, dtype: object


     age sex  occupation zip_code
1     24   M  technician    85711
50    21   M      writer    52245
300   26   F  programmer    55106
```

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)
print users.head()
   user_id  age sex  occupation zip_code
0        1   24   M  technician    85711
1        2   53   F       other    94043
2        3   23   M      writer    32067
3        4   24   M  technician    43537
4        5   33   F       other    15213
```

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

1.3 Joining

Throughout an analysis, we'll often need to merge/join datasets as data is 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 outer join) right: use only keys from right frame (SQL: right outer join) outer: use union of keys from both frames (SQL: full outer join) inner: use intersection of keys from both frames (SQL: inner join) Below are some examples of what each look like.

```
left_frame = pd.DataFrame({'key': range(5),
                           'left_value': ['a', 'b', 'c', 'd', 'e']})
right_frame = pd.DataFrame({'key': range(2, 7),
                           'right_value': ['f', 'g', 'h', 'i', 'j']})
print left_frame
print '\n'
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
```

1.4 inner join (default)

```
print pd.merge(left_frame, right_frame, on='key', how='inner')
   key left_value right_value
0    2          c           f
1    3          d           g
2    4          e           h
```

We lose values from both frames since certain keys do not match up. The SQL equivalent is:

```
SELECT left_frame.key, left_frame.left_value, right_frame.right_value
FROM left_frame
INNER JOIN right_frame
ON left_frame.key = right_frame.key;
```