**Exploring Data**

Exploring probes deeper into the realm of data. An important topic in data science is dimensionality reduction. This chapter borrows munged data from Chapter 5 to demonstrate how this works. Another topic is speed simulation. When working with large datasets, speed is of great importance. Big data is explored with a popular dataset used by academics and industry. Finally, Twitter and Web scraping are two important data sources for exploration.

1. Heat maps
2. Principal component analysis
3. Speed simulation
4. Big data
5. Twitter
6. Web scraping

*Heat Maps*

Heat maps were introduced in chapter 5, but one wasn’t created for the munged dataset. So, we start by creating a Heat map visualization of the wrangled.json data.

import json, pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

def read\_json(f):

with open(f) as f:

return json.load(f)

def verify\_keys(d, \*\*kwargs):

data = d[0].items()

k1 = set([tup[0] for tup in data])

s = kwargs.items()

k2 = set([tup[1] for tup in s])

return list(k1.intersection(k2))

def build\_ls(k, d):

return [{k: row[k] for k in (keys)} for row in d]

def get\_rows(d, n):

[print(row) for i, row in enumerate(d) if i < n]

def conv\_float(d):

return [dict([k, float(v)] for k, v in row.items()) for row in d]

if \_\_name\_\_ == "\_\_main\_\_":

f= 'data/wrangled.json'

data = read\_json(f)

keys = verify\_keys(data, c1='sale', c2='quan', c3='disc', c4='prof')

heat = build\_ls(keys, data)

print ('1st row in "heat":')

get\_rows(heat, 1)

heat = conv\_float(heat)

print ('\n1st row in "heat" converted to float:')

get\_rows(heat, 1)

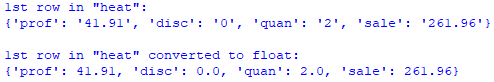
df = pd.DataFrame(heat)

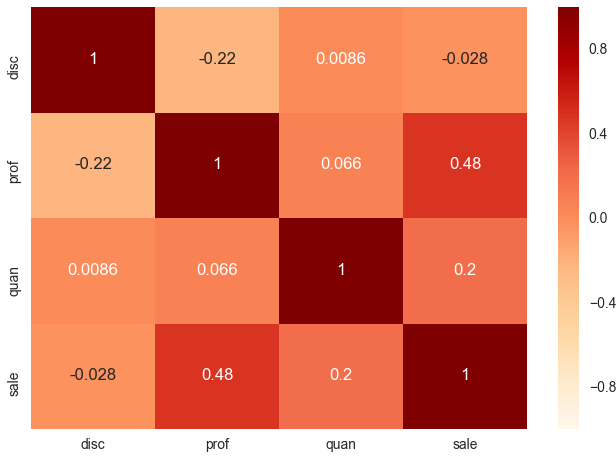
plt.figure()

sns.heatmap(df.corr(), annot=True, cmap='OrRd')

plt.show()

Output:





The code example begins by importing json, pandas, matplotlib, and seaborn libraries. Function read\_json() reads a JSON file. Function verify\_keys() ensures that the keys of interest exist in the JSON file. This is important because we can only create a Heat map based on numerical variables, and the only candidates from the JSON file are sales, quantity, discount, and profit. Function build\_ls() builds a list of dictionary elements based on the numerical variables. Function get\_rows() returns n rows from a list. Function conv\_float() converts dictionary elements to float. The main block begins by reading JSON file wrangled.json. It continues by getting keys for only numerical variables. Next, it builds list a list of dictionary elements (heat) based on the appropriate keys. The code displays the 1st row in heat to verify that all values are float. Since they are not, the code converts them to float. The code then creates a df from heat and plots the Heat map.

*Principal Component Analysis*

Principal Component Analysis (PCA) finds the principal components of data. Principal components represent the underlying structure in the data because they uncover the directions where the data has the most variance (most spread out). PCA leverages eigenvectors and eigenvalues to uncover data variance. An eigenvector is a direction, while an eigenvalue is a number that indicates variance (in the data) in the direction of the eigenvector. The eigenvector with the highest eigenvalue is the principal component. A dataset can be deconstructed into eigenvectors and eigenvalues. The amount of eigenvectors (and eigenvalues) in a dataset equals the number of dimensions. Since the wrangled.json dataset has 4 dimensions (variables), it has 4 eigenvectors/eigenvalues.

The 1st code example runs PCA on the wrangled.json dataset. However, PCA only works with numeric data, so the dataset is distilled down to only those features.

import matplotlib.pyplot as plt, pandas as pd

import numpy as np, json, random as rnd

from sklearn.preprocessing import StandardScaler

from pandas.plotting import parallel\_coordinates

def read\_json(f):

with open(f) as f:

return json.load(f)

def unique\_features(k, d):

return list(set([dic[k] for dic in d]))

def sire\_features(k, d):

return [{k: row[k] for k in (k)} for row in d]

def sire\_numeric(k, d):

s = conv\_float(sire\_features(k, d))

return s

def sire\_sample(k, v, d, m):

indices = np.arange(0, len(d), 1)

s = [d[i] for i in indices if d[i][k] == v]

n = len(s)

num\_keys = ['sale', 'quan', 'disc', 'prof']

for i, row in enumerate(s):

for k in num\_keys:

row[k] = float(row[k])

s = rnd\_sample(m, len(s), s)

return (s, n)

def rnd\_sample(m, n, d):

indices = sorted(rnd.sample(range(n), m))

return [d[i] for i in indices]

def conv\_float(d):

return [dict([k, float(v)] for k, v in row.items()) for row in d]

if \_\_name\_\_ == "\_\_main\_\_":

f = 'data/wrangled.json'

data = read\_json(f)

segm = unique\_features('segm', data)

print ('classes in "segm" feature:')

print (segm)

keys = ['sale', 'quan', 'disc', 'prof', 'segm']

features = sire\_features(keys, data)

num\_keys = ['sale', 'quan', 'disc', 'prof']

numeric\_data = sire\_numeric(num\_keys, features)

k, v = "segm", "Home Office"

m = 100

s\_home = sire\_sample(k, v, features, m)

v = "Consumer"

s\_cons = sire\_sample(k, v, features, m)

v = "Corporate"

s\_corp = sire\_sample(k, v, features, m)

print ('\nHome Office slice:', s\_home[1])

print('Consumer slice:', s\_cons[1])

print ('Coporate slice:', s\_corp[1])

print ('sample size:', m)

df\_home = pd.DataFrame(s\_home[0])

df\_cons = pd.DataFrame(s\_cons[0])

df\_corp = pd.DataFrame(s\_corp[0])

frames = [df\_home, df\_cons, df\_corp]

result = pd.concat(frames)

plt.figure()

parallel\_coordinates(result, 'segm', color=

['orange','lime','fuchsia'])

df = pd.DataFrame(numeric\_data)

X = df.ix[:].values

X\_std = StandardScaler().fit\_transform(X)

mean\_vec = np.mean(X\_std, axis=0)

cov\_mat = np.cov(X\_std.T)

print ('\ncovariance matrix:\n', cov\_mat)

eig\_vals, eig\_vecs = np.linalg.eig(cov\_mat)

print ('\nEigenvectors:\n', eig\_vecs)

print ('\nEigenvalues:\n', np.sort(eig\_vals)[::-1])

tot = sum(eig\_vals)

var\_exp = [(i / tot)\*100 for i in sorted(eig\_vals, reverse=True)]

print ('\nvariance explained:\n', var\_exp)

corr\_mat = np.corrcoef(X.T)

print ('\ncorrelation matrix:\n', corr\_mat)

eig\_vals, eig\_vecs = np.linalg.eig(corr\_mat)

print ('\nEigenvectors:\n', eig\_vecs)

print ('\nEigenvalues:\n', np.sort(eig\_vals)[::-1])

tot = sum(eig\_vals)

var\_exp = [(i / tot)\*100 for i in sorted(eig\_vals, reverse=True)]

print ('\nvariance explained:\n', var\_exp)

cum\_var\_exp = np.cumsum(var\_exp)

fig, ax = plt.subplots()

labels = ['PC1', 'PC2', 'PC3', 'PC4']

width = 0.35

index = np.arange(len(var\_exp))

ax.bar(index, var\_exp,

color=['fuchsia', 'lime', 'thistle', 'thistle'])

for i, v in enumerate(var\_exp):

v = round(v, 2)

val = str(v) + '%'

ax.text(i, v+0.5, val, ha='center', color='b',

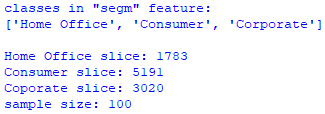
fontsize=9, fontweight='bold')

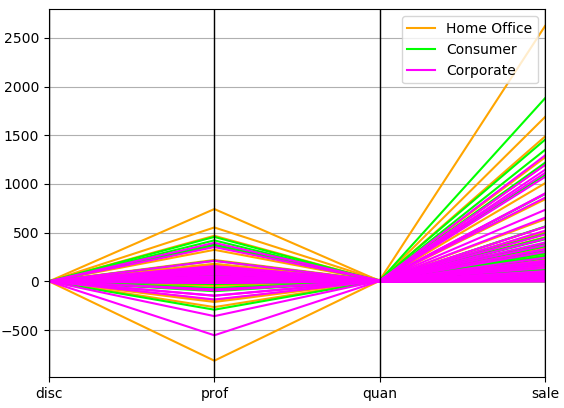
plt.xticks(index, labels)

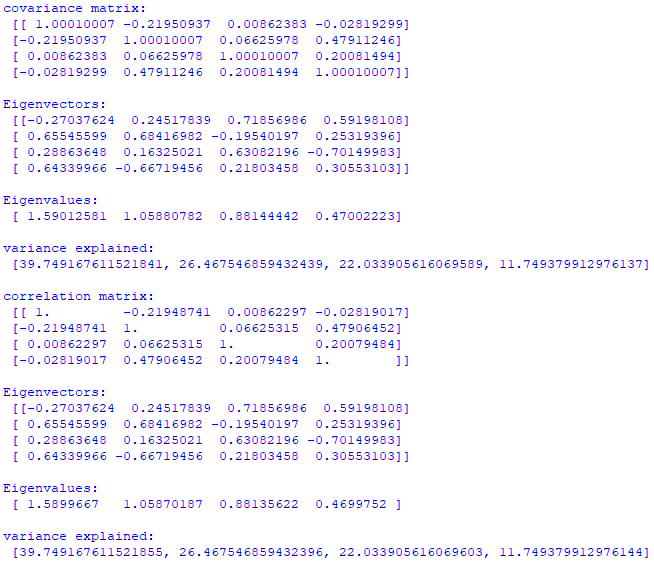
plt.title('Variance Explained')

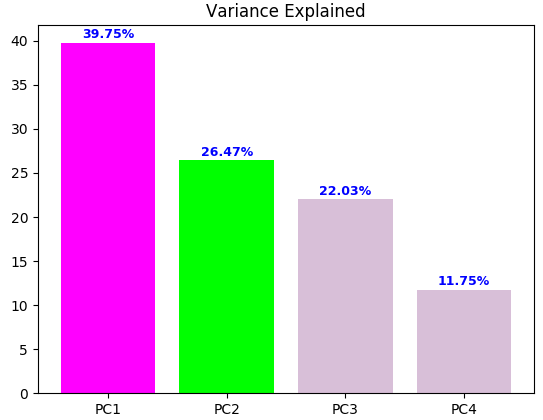
plt.show()

Output:









The code example begins by importing matplotlib, pandas, numpy, json, random, and sklearn libraries. Function read\_json() reads a JSON file. Function unique\_features() distills unique categories (classes) from a dimension (feature). In this case, it distills three classes – Home Office, Corporate, and Consumer – from the segm feature. Since the dataset is close to 10,000 records, I wanted to be sure what classes are in it. Function sire\_features() distills a new dataset with only features of interest. Function sire\_numeric() converts numeric strings to float. Function sire\_sample() returns a random sample of n records filtered for a class. Function rnd\_sample() creates a random sample. Function convert\_float() converts numeric string data to float.

The main block begins by reading wrangled.json and creating dataset features with only features of interest. The code continues by creating dataset numeric that only includes features with numeric data. Dataset numeric is used to generate PCA. Next, three samples of size 100 are created; one for each class. The samples are used to create the parallel coordinates visualization. Code for PCA follows by standardizing and transforming the numeric dataset. A covariance matrix is created so that Eigenvectors and Eigenvalues can be generated. I include PCA using the correlation matrix because some disciplines prefer it. Finally, a visualization of the principal components is created.

Parallel coordinates show that prof (profit) and sale (sales) are the most important features. The PCA visualization (Variance Explained) shows that the 1st principal component accounts for 39.75%, 2nd 26.47%, 3rd 22.03%, and 4th 11.75%. PCA analysis is not very useful in this case since all four principal components are necessary, especially the 1st three. So, we cannot drop any of the dimensions from future analysis.

The 2nd code example uses the iris dataset for PCA:

import matplotlib.pyplot as plt, pandas as pd, numpy as np

from sklearn.preprocessing import StandardScaler

from pandas.plotting import parallel\_coordinates

def conv\_float(d):

return d.astype(float)

if \_\_name\_\_ == "\_\_main\_\_":

df = pd.read\_csv('data/iris.csv')

X = df.ix[:,0:4].values

y = df.ix[:,4].values

X\_std = StandardScaler().fit\_transform(X)

mean\_vec = np.mean(X\_std, axis=0)

cov\_mat = np.cov(X\_std.T)

eig\_vals, eig\_vecs = np.linalg.eig(cov\_mat)

print ('Eigenvectors:\n', eig\_vecs)

print ('\nEigenvalues:\n', eig\_vals)

plt.figure()

parallel\_coordinates(df, 'Name', color=

['orange','lime','fuchsia'])

tot = sum(eig\_vals)

var\_exp = [(i / tot)\*100 for i in sorted(eig\_vals, reverse=True)]

cum\_var\_exp = np.cumsum(var\_exp)

fig, ax = plt.subplots()

labels = ['PC1', 'PC2', 'PC3', 'PC4']

width = 0.35

index = np.arange(len(var\_exp))

ax.bar(index, var\_exp,

color=['fuchsia', 'lime', 'thistle', 'thistle'])

for i, v in enumerate(var\_exp):

v = round(v, 2)

val = str(v) + '%'

ax.text(i, v+0.5, val, ha='center', color='b',

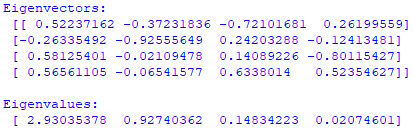
fontsize=9, fontweight='bold')

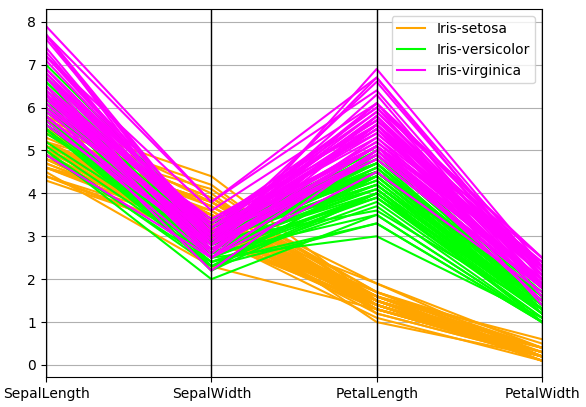
plt.xticks(index, labels)

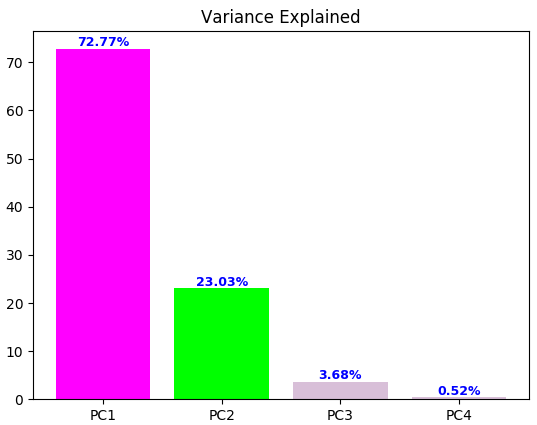
plt.title('Variance Explained')

plt.show()

Output:







The code example is much shorter than the previous one because we didn’t have to wrangle, clean (as much), and create random samples (for Parallel Coordinates visualization). The code begins by importing matplotlib, pandas, numpy, and sklearn libraries. Function conv\_float() converts numeric strings to float. The main block begins by reading the iris dataset. It continues by standardizing and transforming the data for PCA. Parallel Coordinates and variance explained are then displayed.

Parallel Coordinates shows that PetalLength and PetalWidth are the most important features. The PCA visualization (Variance Explained) shows that the 1st principal component accounts for 72.77%, 2nd 23.03%, 3rd 3.68%, and 4th 0.52%. PCA analysis is very useful in this case because the 1st two principal components account for over 95% of the variance. So, we can drop PC3 and PC4 from further consideration.

For clarity, the 1st step for PCA is to explore the Eigenvectors and Eigenvalues. The Eigenvectors with the lowest Eigenvalues bear the least information about the distribution of the data, so they can be dropped. In this example, the 1st two Eigenvalues are much higher, especially PC1. Dropping PC3 and PC4 are thereby in order. The 2nd step is to measure explained variance, which can be calculated from the eigenvalues. Explained variance tells us how much information (variance) can be attributed to each of the principal components. Looking at explained variance confirms that PC3 and PC4 are not important.

*Speed Simulation*

Speed in data science is important, especially as datasets become bigger. Generators are helpful in memory optimization because a generator function returns one item at a time rather than all items at once.

The code example contrasts speed between a list and a generator:

import json, humanfriendly as hf

from time import clock

def read\_json(f):

with open(f) as f:

return json.load(f)

def mk\_gen(k, d):

for row in d:

dic = {}

for key in k:

dic[key] = float(row[key])

yield dic

def conv\_float(keys, d):

return [dict([k, float(v)] for k, v in row.items()

if k in keys) for row in d]

if \_\_name\_\_ == "\_\_main\_\_":

f = 'data/wrangled.json'

data = read\_json(f)

keys = ['sale', 'quan', 'disc', 'prof']

print ('create, convert, and display list:')

start = clock()

data = conv\_float(keys, data)

for i, row in enumerate(data):

if i < 5:

print (row)

end = clock()

elapsed\_ls = end - start

print (hf.format\_timespan(elapsed\_ls, detailed=True))

print ('\ncreate, convert, and display generator:')

start = clock()

generator = mk\_gen(keys, data)

for i, row in enumerate(generator):

if i < 5:

print (row)

end = clock()

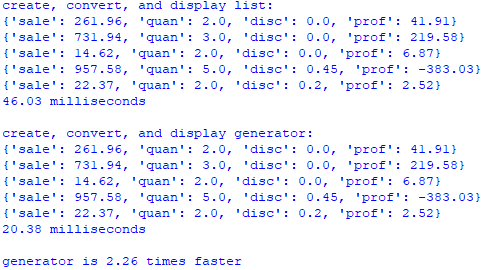
elapsed\_gen = end - start

print (hf.format\_timespan(elapsed\_gen, detailed=True))

speed = round(elapsed\_ls / elapsed\_gen, 2)

print ('\ngenerator is', speed, 'times faster')

Output:



The code example begins by importing json, humanfriendly, and time libraries. You may have to install humanfriendly like I did. Function read\_json() reads JSON. Function mk\_gen() creates a generator based on four features from wrangled.json and converts values to float. Function conv\_float() converts dictionary values from a list to float. The main block begins by reading wrangled.json into a list. The code continues by timing the process of creating a new list from keys and converting values to float. Next, a generator is created that mimics the list creating and conversion process. The generator is 2.26 times faster.

*Big Data*

Big data is the rage of the 21st century. So, let’s work with a relatively big dataset. GroupLens is a website that offers access to large social computing datasets for theory and practice. GroupLens has collected and made available rating datasets from the MovieLens website:

<https://grouplens.org/datasets/movielens/>. We are going to explore the 1M dataset, which contains approximately 1 million ratings from 6 thousand users on 4 thousand movies. I was hesitant to wrangle, cleanse, and process a dataset over 1 million because of the limited processing power of my relatively new PC.

The 1st code example reads, cleans, sizes, and dumps MovieLens data to JSON:

import json, csv

def read\_dat(h, f):

return csv.DictReader((line.replace('::', ':')

for line in open(f)),

delimiter=':', fieldnames=h,

quoting=csv.QUOTE\_NONE)

def gen\_dict(d):

for row in d:

yield dict(row)

def dump\_json(f, l, d):

f = open(f, 'w')

f.write('[')

for i, row in enumerate(d):

j = json.dumps(row)

f.write(j)

if i < l - 1:

f.write(',')

else:

f.write(']')

f.close()

def read\_json(f):

with open(f) as f:

return json.load(f)

def display(n, f):

for i, row in enumerate(f):

if i < n:

print (row)

print()

if \_\_name\_\_ == "\_\_main\_\_":

print ('... sizing data ...\n')

u\_dat = 'data/ml-1m/users.dat'

m\_dat = 'data/ml-1m/movies.dat'

r\_dat = 'data/ml-1m/ratings.dat'

unames = ['user\_id', 'gender', 'age', 'occupation', 'zip']

mnames = ['movie\_id', 'title', 'genres']

rnames = ['user\_id', 'movie\_id', 'rating', 'timestamp']

users = read\_dat(unames, u\_dat)

ul = len(list(gen\_dict(users)))

movies = read\_dat(mnames, m\_dat)

ml = len(list(gen\_dict(movies)))

ratings = read\_dat(rnames, r\_dat)

rl = len(list(gen\_dict(ratings)))

print ('size of datasets:')

print ('users', ul)

print ('movies', ml)

print ('ratings', rl)

print ('\n... dumping data ...\n')

users = read\_dat(unames, u\_dat)

users = gen\_dict(users)

movies = read\_dat(mnames, m\_dat)

movies = gen\_dict(movies)

ratings = read\_dat(rnames, r\_dat)

ratings = gen\_dict(ratings)

uf = 'data/users.json'

dump\_json(uf, ul, users)

mf = 'data/movies.json'

dump\_json(mf, ml, movies)

rf = 'data/ratings.json'

dump\_json(rf, rl, ratings)

print ('\n... verifying data ...\n')

u = read\_json(uf)

m = read\_json(mf)

r = read\_json(rf)

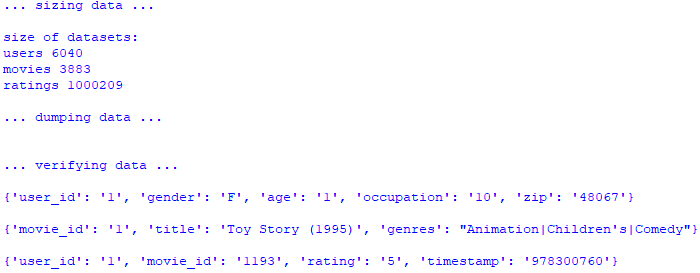
n = 1

display(n, u)

display(n, m)

display(n, r)

Output:



The code example begins by importing json and csv libraries. Function read\_dat() reads and cleans the data (replaces double colons with single colons as delimiters). Function gen\_dict() converts an OrderedDict list to a regular dictionary list for easier processing. Function dump\_json() is a custom function that I wrote to dump data to JSON. Function read\_json() reads JSON. Function display() displays some data for verification. The main block begins by reading the three datasets and finding their sizes. It continues by rereading the datasets and dumping to JSON. The datasets need to be reread because a generator can only be traversed once. Since the ratings dataset is over 1 million records, it takes a few seconds to process.

The 2nd code example cleans the movie dataset, which requires extensive additional cleaning:

import json, numpy as np

def read\_json(f):

with open(f) as f:

return json.load(f)

def dump\_json(f, d):

with open(f, 'w') as fout:

json.dump(d, fout)

def display(n, d):

[print (row) for i,row in enumerate(d) if i < n]

def get\_indx(k, d):

return [row[k] for row in d if 'null' in row]

def get\_data(k, l, d):

return [row for i, row in enumerate(d) if row[k] in l]

def get\_unique(key, d):

s = set()

for row in d:

for k, v in row.items():

if k in key:

s.add(v)

return np.sort(list(s))

if \_\_name\_\_ == "\_\_main\_\_":

mf = 'data/movies.json'

m = read\_json(mf)

n = 20

display(n, m)

print ()

indx = get\_indx('movie\_id', m)

for row in m:

if row['movie\_id'] in indx:

row['title'] = row['title'] + ':' + row['genres']

row['genres'] = row['null'][0]

del row['null']

title = row['title'].split(" ")

year = title.pop()

year = ''.join(c for c in year if c not in '()')

row['title'] = ' '.join(title)

row['year'] = year

data = get\_data('movie\_id', indx, m)

n = 2

display(n, data)

s = get\_unique('year', m)

print ('\n', s, '\n')

rec = get\_data('year', ['Assignment'], m)

print (rec[0])

rec = get\_data('year', ["L'Associe1982"], m)

print (rec[0], '\n')

b1, b2, cnt = False, False, 0

for row in m:

if row['movie\_id'] in ['1001']:

row['year'] = '1982'

print (row)

b1 = True

elif row['movie\_id'] in ['2382']:

row['title'] = 'Police Academy 5: Assignment: Miami Beach'

row['genres'] = 'Comedy'

row['year'] = '1988'

print (row)

b2 = True

elif b1 and b2: break

cnt += 1

print ('\n', cnt, len(m))

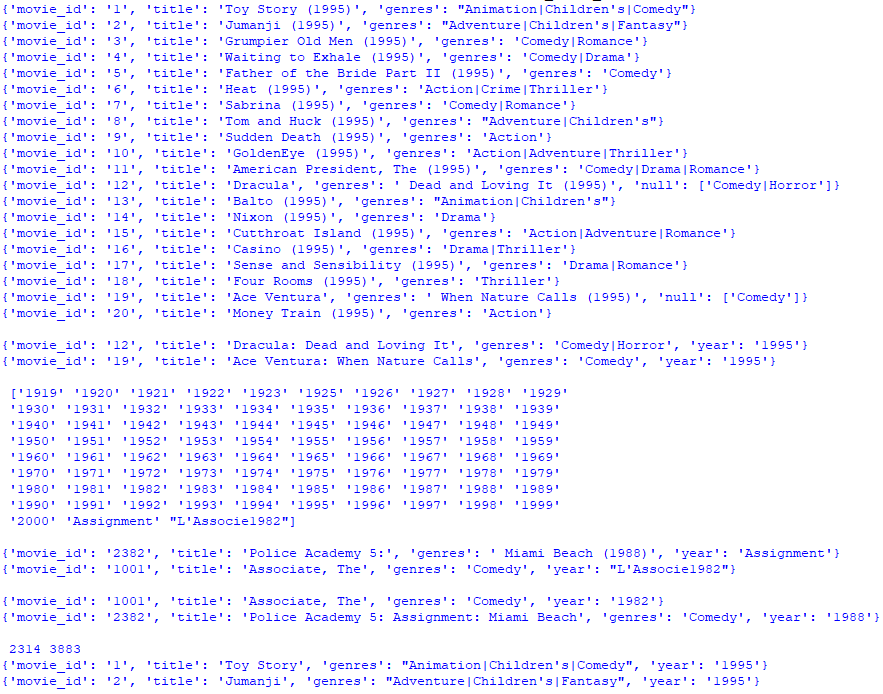
mf = 'data/cmovies.json'

dump\_json(mf, m)

m = read\_json(mf)

display(n, m)

Output:



The code example begins by importing json and numpy libraries. Function read\_json() reads JSON. Function dump\_json() saves JSON. Function display() displays n records. Function get\_indx() returns indices of dictionary elements with a null key. Function get\_data() returns a dataset filtered by indices and movie\_id key. Function get\_unique() returns a list of unique values from a list of dictionary elements. The main block begins by reading movies.json and displaying for inspection. Records 12 and 19 have a null key. The code continues by finding all movie\_id indices with a null key. The next several lines clean all movies. Those with a null key require added logic to fully clean, but all records have modified titles and a new year key. To verify, records 12 and 19 are displayed. To be sure that all is well, the code finds all unique keys based on year. Notice that there are two records that don’t have a legitimate year. So, the code cleans the two records. The 2nd elif was added to the code to stop processing once the two dirty records were cleaned. Although not included in the code, I checked movie\_id, title, and genres keys but found no issues.

The 3rd code example generates useful information from the three datasets:

import json, numpy as np, sys, os, humanfriendly as hf

from time import clock

sys.path.append(os.getcwd()+'/classes')

import conn

def read\_json(f):

with open(f) as f:

return json.load(f)

def get\_column(A, v):

return [A\_i[v] for A\_i in A]

def remove\_nr(v1, v2):

set\_v1 = set(v1)

set\_v2 = set(v2)

diff = list(set\_v1 - set\_v2)

return diff

def get\_info(\*args):

a = [arg for arg in args]

ratings = [int(row[a[0][1]]) for row in a[2] if row[a[0][0]] == a[1]]

uids = [row[a[0][3]] for row in a[2] if row[a[0][0]] == a[1]]

title = [row[a[0][2]] for row in a[3] if row[a[0][0]] == a[1]]

age = [int(row[a[0][4]]) for col in uids for row in a[4] if col == row[a[0][3]]]

gender = [row[a[0][5]] for col in uids for row in users if col == row[a[0][3]]]

return (ratings, title[0], uids, age, gender)

def generate(k, v, r, m, u):

for i, mid in enumerate(v):

dic = {}

rec = get\_info(k, mid, r, m, u)

dic = {'\_id':i, 'mid':mid, 'title':rec[1], 'avg\_rating':np.mean(rec[0]),

'n\_ratings':len(rec[0]), 'avg\_age':np.mean(rec[3]),

'M':rec[4].count('M'), 'F':rec[4].count('F')}

dic['avg\_rating'] = round(float(str(dic['avg\_rating'])[:6]),2)

dic['avg\_age'] = round(float(str(dic['avg\_age'])[:6]))

yield dic

def gen\_ls(g):

for i, row in enumerate(g):

yield row

if \_\_name\_\_ == "\_\_main\_\_":

print ('... creating datasets ...\n')

m = 'data/cmovies.json'

movies = np.array(read\_json(m))

r = 'data/ratings.json'

ratings = np.array(read\_json(r))

r = 'data/users.json'

users = np.array(read\_json(r))

print ('... creating movie indicies vector data ...\n')

mv = get\_column(movies, 'movie\_id')

rv = get\_column(ratings, 'movie\_id')

print ('... creating unrated movie indicies vector ...\n')

nrv = remove\_nr(mv, rv)

diff = [int(row) for row in nrv]

print (np.sort(diff), '\n')

new\_mv = [x for x in mv if x not in nrv]

mid = '1'

keys = ('movie\_id', 'rating', 'title', 'user\_id', 'age', 'gender')

stats = get\_info(keys, mid, ratings, movies, users)

avg\_rating = np.mean(stats[0])

avg\_age = np.mean(stats[3])

n\_ratings = len(stats[0])

title = stats[1]

M, F = stats[4].count('M'), stats[4].count('F')

print ('avg rating for:', end=' "')

print (title + '" is', round(avg\_rating, 2), end=' (')

print (n\_ratings, 'ratings)\n')

gen = generate(keys, new\_mv, ratings, movies, users)

gls = gen\_ls(gen)

obj = conn.conn('test')

db = obj.getDB()

movie\_info = db.movie\_info

movie\_info.drop()

print ('... saving movie\_info to MongoDB ...\n')

start = clock()

for row in gls:

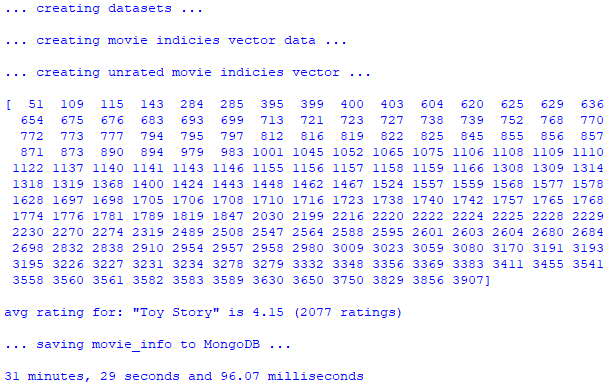
movie\_info.insert(row)

end = clock()

elapsed\_ls = end - start

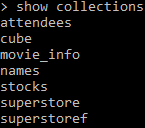
print (hf.format\_timespan(elapsed\_ls, detailed=True))

Output:



The code example begins by importing json, numpy, sys, os, humanfriendly, time, and conn (a custom class I created to connect to MongoDB). Function read\_json() reads JSON. Function get\_column() returns a column vector. Function remove\_nr() removes movie\_id values that are not rated. Function get\_info() returns ratings, users, age, and gender as column vectors as well as title of a movie. The function is very complex because each vector is created by traversing one of the data sets and making comparisons. To make it more concise, list comprehension was used extensively. Function generate() generates a dictionary element that contains average rating, average age, number of males and females raters, number of ratings, movie\_id, and title of each movie. Function gen\_ls() generates each dictionary element generated by function generate(). The main block begins by reading the three JSON datasets. It continues by getting two column vectors – each movie\_id from movies dataset and movie\_id from ratings dataset. Each column vector is converted to a set to remove duplicates. Column vectors are used instead of full records for faster processing. Next, a new column vector is returned containing only movies that are rated. The code continues by getting title and column vectors for ratings, and users, age, and gender for each movie with movie\_id of 1. The average rating for this movie is displayed with its title and number of ratings. The final part of the code creates a generator containing a list of dictionary elements. Each dictionary element contains the movie\_id, title, average rating, average age, number of ratings, number of male raters, and number of female raters. Next, another generator is created to generate the list. Creating the generators is instantaneous, but unraveling (unfolding) contents takes time. Keep in mind that the 1st generator runs billions of processes and 2nd generator runs the 1st one. So, saving contents to MongoDB takes close to half an hour.

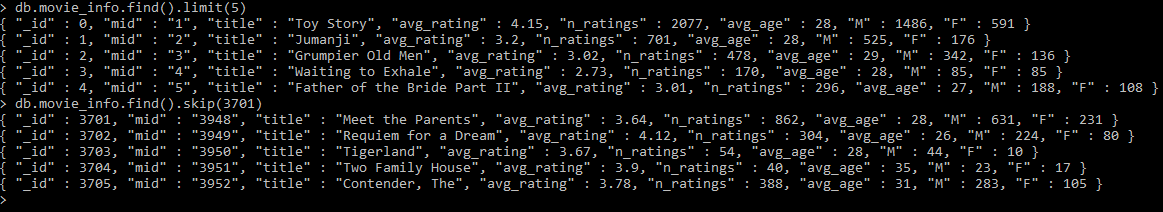
To verify results, let’s look at the data in MongoDB. The command show collections is the 1st that I run to check if collection movie\_info was created:



Next, I run db.movie\_info.count() to check the number of documents:



Now that I know the number of documents, I can display the 1st and last five records:



From data exploration, it appears that the movie\_info collection was created correctly.

The 4th code example saves the three datasets – users.json, cmovies.json, and ratings.json – to MongoDB:

import sys, os, json, humanfriendly as hf

from time import clock

sys.path.append(os.getcwd() + '/classes')

import conn

def read\_json(f):

with open(f) as f:

return json.load(f)

def create\_db(c, d):

c = db[c]

c.drop()

for i, row in enumerate(d):

row['\_id'] = i

c.insert(row)

if \_\_name\_\_ == "\_\_main\_\_":

u = read\_json('data/users.json')

m = read\_json('data/cmovies.json')

r = read\_json('data/ratings.json')

obj = conn.conn('test')

db = obj.getDB()

print ('... creating MongoDB collections ...\n')

start = clock()

create\_db('users', u)

create\_db('movies', m)

create\_db('ratings', r)

end = clock()

elapsed\_ls = end - start

print (hf.format\_timespan(elapsed\_ls, detailed=True))

Output:



The code example begins by importing sys, os, json, humanfriendly, time, and custom class conn. Function read\_json reads JSON. Function create\_db() creates MongoDB collections. The main block begins by reading the three datasets – users.json, cmovies.json, and ratings.json – and saving them to MongoDB collections. Since the ratings.json dataset is over 1 million records, it takes some time to save it to the database.

The 5th code example introduces the aggregation pipeline, which is a MongoDB framework for data aggregation modeled on the concept of data processing pipelines. Documents enter a multi-stage pipeline that transforms them into aggregated results. In addition to grouping and sorting documents by specific field or fields and aggregating contents of arrays, pipeline stages can use operators for tasks such as calculating averages or concatenating strings. The pipeline provides efficient data aggregation using native MongoDB operations, and is the preferred method for data aggregation in MongoDB.

import sys, os

sys.path.append(os.getcwd() + '/classes')

import conn

def match\_item(k, v, d):

pipeline = [ {'$match' : { k : v }} ]

q = db.command('aggregate',d,pipeline=pipeline)

return q

if \_\_name\_\_ == "\_\_main\_\_":

obj = conn.conn('test')

db = obj.getDB()

movie = 'Toy Story'

q = match\_item('title', movie, 'movie\_info')

r = q['result'][0]

print (movie, 'document:')

print (r)

print ('average rating', r['avg\_rating'], '\n')

user\_id = '3'

print ('\*\*\* user', user\_id, '\*\*\*')

q = match\_item('user\_id', user\_id, 'users')

r = q['result'][0]

print ('age', r['age'], 'gender', r['gender'], 'occupation',\

r['occupation'], 'zip', r['zip'], '\n')

print ('\*\*\* "user 3" movie ratings of 5 \*\*\*')

q = match\_item('user\_id', user\_id, 'ratings')

mid = q['result']

for row in mid:

if row['rating'] == '5':

q = match\_item('movie\_id', row['movie\_id'], 'movies')

title = q['result'][0]['title']

genre = q['result'][0]['genres']

print (row['movie\_id'], title, genre)

mid = '1136'

q = match\_item('mid', mid, 'movie\_info')

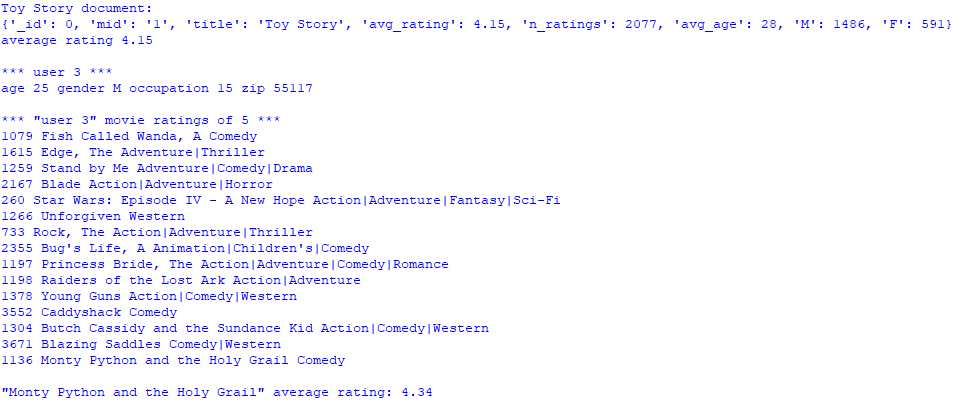
title = q['result'][0]['title']

avg\_rating = q['result'][0]['avg\_rating']

print ()

print ('"' + title + '"', 'average rating:', avg\_rating)

Output:



The code example begins by importing sys, os, and custom class conn. Function match\_item() uses the aggregation pipeline to match records to criteria. The main block begins by using the aggregation pipeline to return the Toy Story document from collection movie\_info. The code continues by using the pipeline to return the user 3 document from collection users. Next, the aggregation pipeline is used to return all movie ratings of 5 for user 3. Finally, the pipeline is used to return the average rating for Monty Python and the Holy Grail from collection movie\_info. The aggregation pipeline is efficient and offers a vast array of functionality.

The 6th code example demonstrates a multi-stage aggregation pipeline:

import sys, os

sys.path.append(os.getcwd() + '/classes')

import conn

def stages(k, v, r, d):

pipeline = [ {'$match' : { '$and' : [ { k : v },

{'rating':{'$eq':r} }] } },

{'$project' : {

'\_id' : 1,

'user\_id' : 1,

'movie\_id' : 1,

'rating' : 1 } },

{'$limit' : 100}]

q = db.command('aggregate',d,pipeline=pipeline)

return q

def match\_item(k, v, d):

pipeline = [ {'$match' : { k : v }} ]

q = db.command('aggregate',d,pipeline=pipeline)

return q

if \_\_name\_\_ == "\_\_main\_\_":

obj = conn.conn('test')

db = obj.getDB()

u = '3'

r = '5'

q = stages('user\_id', u, r, 'ratings')

result = q['result']

print ('ratings of', r, 'for user ' + str(u) + ':')

for i, row in enumerate(result):

print (row)

n = i+1

print ()

print (n, 'associated movie titles:')

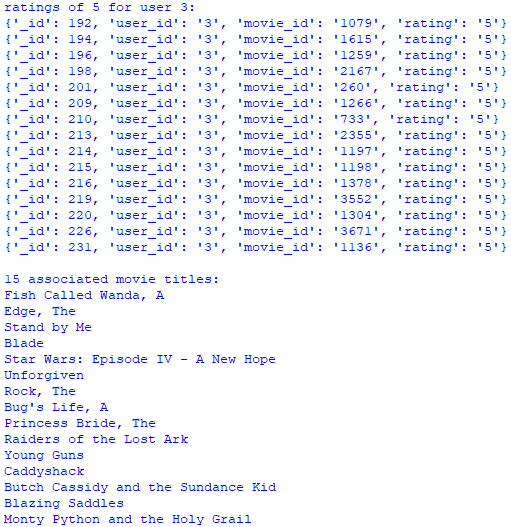
for i, row in enumerate(result):

q = match\_item('movie\_id', row['movie\_id'], 'movies')

r = q['result'][0]

print (r['title'])

Output:



The code example begins by importing sys, os, and custom class conn. Function stages() uses a 3-stage aggregation pipeline. The 1st stage finds all ratings of 5 from user 3. The 2nd stage projects the fields to be displayed. The 3rd stage limits the number of documents returned. It is important to include a limit stage because the results database is big and pipelines have size limitations. Function match\_item() uses the aggregation pipeline to match records to criteria. The main block begins by using the stages() pipeline to return all ratings of 5 from user 3. The code continues by iterating this data and using the match\_item() pipeline to get the titles that user 3 rated as 5. The pipeline is an efficient method to query documents from MongoDB, but takes practice to get acquainted with its syntax.

*Twitter*

Twitter is a fantastic source of data because you can get data about almost anything. To access data from Twitter, you need to connect to the Twitter Streaming API. Connection requires 4 pieces of information from Twitter – API key, API secret, Access token, and Access token secret (encrypted). After you register and get your credentials, you need to install a Twitter API. I chose the Twitter API TwitterSearch, but there are many others.

The 1st code example creates JSON to hold my Twitter credentials (insert your credentials into each variable):

import json

if \_\_name\_\_ == '\_\_main\_\_':

consumer\_key = ''

consumer\_secret = ''

access\_token = ''

access\_encrypted = ''

data = {}

data['ck'] = consumer\_key

data['cs'] = consumer\_secret

data['at'] = access\_token

data['ae'] = access\_encrypted

json\_data = json.dumps(data)

header = '[\n'

ender = ']'

obj = open('data/credentials.json', 'w')

obj.write(header)

obj.write(json\_data + '\n')

obj.write(ender)

obj.close()

I chose to save credentials in JSON to hide them from view. The code example imports the json library. The main block saves credentials into JSON.

The 2nd code example streams Twitter data using the TwitterSearch API:

from TwitterSearch import \*

import json, sys

class twitSearch:

def \_\_init\_\_(self, cred, ls, limit):

self.cred = cred

self.ls = ls

self.limit = limit

def search(self):

num = 0

dt = []

dic = {}

try:

tso = TwitterSearchOrder()

tso.set\_keywords(self.ls)

tso.set\_language('en')

tso.set\_include\_entities(False)

ts = TwitterSearch(

consumer\_key = self.cred[0]['ck'],

consumer\_secret = self.cred[0]['cs'],

access\_token = self.cred[0]['at'],

access\_token\_secret = self.cred[0]['ae']

)

for tweet in ts.search\_tweets\_iterable(tso):

if num <= self.limit:

dic['\_id'] = num

dic['tweeter'] = tweet['user']['screen\_name']

dic['tweet\_text'] = tweet['text']

dt.append(dic)

dic = {}

else:

break

num += 1

except TwitterSearchException as e:

print (e)

return dt

def get\_creds():

with open('data/credentials.json') as json\_data:

d = json.load(json\_data)

json\_data.close()

return d

def write\_json(f, d):

with open(f, 'w') as fout:

json.dump(d, fout)

def translate():

return dict.fromkeys(range(0x10000, sys.maxunicode + 1), 0xfffd)

def read\_json(f):

with open(f) as f:

return json.load(f)

if \_\_name\_\_ == '\_\_main\_\_':

cred = get\_creds()

ls = ['machine', 'learning']

limit = 10

obj = twitSearch(cred, ls, limit)

data = obj.search()

f = 'data/TwitterSearch.json'

write\_json(f, data)

non\_bmp\_map = translate()

print ('twitter data:')

for row in data:

row['tweet\_text'] = str(row['tweet\_text']).translate(non\_bmp\_map)

tweet\_text = row['tweet\_text'][0:50]

print ('{:<3}{:18s}{}'.format(row['\_id'], row['tweeter'], tweet\_text))

print ('\nverify JSON:')

read\_data = read\_json(f)

for i, p in enumerate(read\_data):

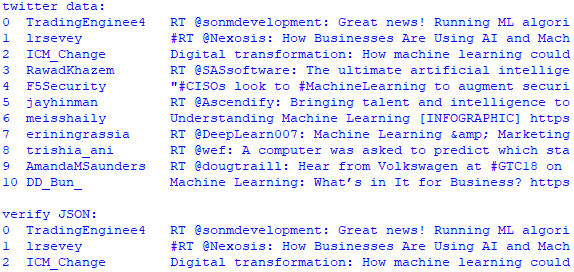
if i < 3:

p['tweet\_text'] = str(p['tweet\_text']).translate(non\_bmp\_map)

tweet\_text = p['tweet\_text'][0:50]

print ('{:<3}{:18s}{}'.format(p['\_id'], p['tweeter'], tweet\_text))

Output:



The code example begins by importing TwitterSearch, json, and sys libraries. Class twitSearch streams Twitter data based on Twitter credentials, a list of keywords, and a limit. Function get\_cred() returns Twitter credentials from JSON. Function write\_json() writes data to JSON. Function translate() converts streamed data outside the Basic Multilingual Plane (BMP) to a usable format. Emoji’s, for example, are outside the BMP. Function read\_json() reads JSON. The main block begins by getting Twitter credentials, creating a list of search keywords, and a limit. In this case, the list of search keywords holds machine and learning because I wanted to stream data about machine learning. Limit of 10 restricts streamed records to 10 tweets. The code continues by writing Twitter data to JSON, translating tweets to control for non-BMP data, and printing the tweet. Finally, the code reads JSON to verify that the tweets were saved properly and prints a few.

*Web Scraping*

Web scraping is a programmatic approach for extracting information from websites. It focuses on transforming unstructured HTML formatted data into structured data. Web scraping is programmatically intensive because of the unstructured nature of HTML. That is, HTML has few if any structural rules, which means that HTML structural patterns tend to differ from one website to another. So, get ready to write custom code for each Web scraping adventure.

The code example scrapes book information from a popular technical book publishing company. The 1st step is to locate the webpage. The 2nd step is to open a window with the source code. The 3rd step is to traverse the source code to identify the data to scrape. The 4th step is to scrape.

With Google Chrome, click More tools and then Developer tools to open the source code window. Next, hover the mouse cursor over the source until you find the data. Move down the source code tree to find the tags you want to scrape. Finally, scrape the data.

from bs4 import BeautifulSoup

import requests, json

def build\_title(t):

t = t.text

t = t.split()

ls = []

for row in t:

if row != '-':

ls.append(row)

elif row == '-':

break

return ' '.join(ls)

def release\_date(r):

r = r.text

r = r.split()

prefix = r[0] + s + r[1]

if len(r) == 5:

date = r[2] + s + r[3] + s + r[4]

else:

date = r[2] + s + r[3]

return prefix, date

def write\_json(f, d):

with open(f, 'w') as fout:

json.dump(d, fout)

def read\_json(f):

with open(f) as f:

return json.load(f)

if \_\_name\_\_ == '\_\_main\_\_':

s = ' '

dic\_ls = []

base\_url = "https://ssearch.oreilly.com/?q=data+science"

soup = BeautifulSoup(requests.get(base\_url).text, 'lxml')

books = soup.find\_all('article')

for i, row in enumerate(books):

dic = {}

tag = row.name

tag\_val = row['class']

title = row.find('p', {'class' : 'title'})

title = build\_title(title)

url = row.find('a', {'class' : 'learn-more'})

learn\_more = url.get('href')

author = row.find('p', {'class' : 'note'}).text

release = row.find('p', {'class' : 'note date2'})

prefix, date = release\_date(release)

if len(tag\_val) == 2:

publisher = row.find('p', {'class' : 'note publisher'}).text

item = row.find('img', {'class' : 'book'})

cat = item.get('class')[0]

else:

publisher, cat = None, None

desc = row.find('p', {'class' : 'description'}).text.split()

desc = [row for i, row in enumerate(desc) if i < 7]

desc = ' '.join(desc) + ' ...'

dic['title'] = title

dic['learn\_more'] = learn\_more

if author[0:3] != 'Pub':

dic['author'] = author

if publisher is not None:

dic['publisher'] = publisher

dic['category'] = cat

else:

dic['event'] = desc

dic['date'] = date

dic\_ls.append(dic)

f = 'data/scraped.json'

write\_json(f, dic\_ls)

data = read\_json(f)

for i, row in enumerate(data):

if i < 6:

print (row['title'])

if 'author' in row.keys():

print (row['author'])

if 'publisher' in row.keys():

print (row['publisher'])

if 'category' in row.keys():

print ('Category:', row['category'])

print ('Release Date:', row['date'])

if 'event' in row.keys():

print ('Event:', row['event'])

print ('Publish Date:', row['date'])

print ('Learn more:', row['learn\_more'])

print ()

Output:



The code example begins by importing BeautifulSoup, request, and json libraries. Function build\_title() builds scraped title data into a string. Function release\_date() builds scraped date data into a string. Function write\_json() and read\_json() write and read JSON respectively. The main block begins by converting the URL page into a BeautifulSoup object. The code continues by placing all article tags into variable books. From exploration, I found that the article tags contained the information I wanted to scrape. Next, each article tag is traversed. Scraping would have been much easier if the information in each article tag was structured consistently. Since it was not, the logic to extract each piece of information is extensive. Each piece of information is placed in a dictionary element, which is subsequently appended to a list. Finally, the list is saved to JSON. The JSON is read and a few records are displayed to verify that all is well.