Python Workshop 6

September 27, 2017

Welcome Back

We've been on hiatus for a few months and most of us can barely remember any Python; or even whether it was Python, Go, JavaScript, Julia, Scala, or something else we were investigating.

This is a good time to update our installation with the latest versions of our favorite packages and maybe even install some new ones. Recall that we used a Python environment manager named **conda** to manage our Python environment. The name comes from the full blown installation called Anaconda (https://docs.continuum.io/) originally developed by Continuum Analytics. **conda** is a minimal installation that allows us to customize the packages we deploy.

Bluecoat

Don't forget that our friend, Bluecoat, hasn't gone anywhere. In order for our **conda** environment manager to reach the package repository, we have to perform a few work-around tasks.

Authenticate Through Firewall

Enter the following URL into your web browser.

https://repo.continuum.io/pkgs/ (https://repo.continuum.io/pkgs/)

Authenticate through the firewall if you have to.

SSL Certificate Verification

Because Bluecoat subverts the server certificate returned from the Conda repository servers, we must tell the conda client to skip the certificate verification step.

```
conda config --set ssl_verify false
```

You can check whether this was already done with

```
conda config --show | grep ssl
```

If your machine doesn't have grep (shame on you), you can manually scroll through the output and verify. It should be near the bottom.

Later we'll learn how to install the Bluecoat certificate into a conda certificate store.

Update Packages

Using

```
conda update <package name>
```

update the following packages.

- `conda` - tell conda to update itself
- `python` - the Python interpreter
- `requests` - high level HTTP package
- `ipython` - the `ipython` shell
- `pandas` - which in turn updates `NumPy` and others
- `jupyter` - interactive web notebooks
- `scipy` - utilities for probability
- `matplotlib` - plotting
- `seaborn` - more plotting
- `statsmodels` - statistical modeling

In [1]:
```
import pandas as pd
df = pd.read_json('https://data.lacounty.gov/resource/uvdj-ch3p.json?$limit=20')
len(df)
```

Out[1]:
```
20
```

We now have a data frame in the `df` variable. We'll have much more to say about invoking APIs over a network later. For now we'll use this opportunity to review basic operations on a data frame.

The `dtypes` attribute tells us the type for each column.

In [35]:
```
df.dtypes
```

Out[35]:
```
city                             object
crime_category_description       object
crime_category_number             int64
crime_date                       object
crime_identifier                  int64
crime_year                        int64
gang_related                     object
geo_location                     object
geo_location_address             object
geo_location_city                object
geo_location_state               object
geo_location_zip                float64
latitude                        float64
longitude                       float64
reporting_district                int64
state                            object
station_identifier               object
station_name                     object
statistical_code                  int64
statistical_code_description     object
street                           object
victim_count                      int64
zip                             float64
dtype: object
```

A dtype of `object` usually means a string. Most of these seem acceptable. A few that stand out are

- `crime_date` - should be a datetime.
- `geo_location_zip` - should be a string.

We can convert this within the `read_json` function call. We'll increase the `$limit=20` parameter so that we receive the default maximum number of rows, which for this API is 1,000.

In [36]:
```
df = pd.read_json('https://data.lacounty.gov/resource/uvdj-ch3p.json?$limit=1000'
,
                  dtype={'geo_location_zip': 'U'}, convert_dates=['crime_date'])
df.dtypes
```

Out[36]:
```
city                                object
crime_category_description          object
crime_category_number                int64
crime_date                  datetime64[ns]
crime_identifier                     int64
crime_year                           int64
gang_related                        object
geo_location                        object
geo_location_address                object
geo_location_city                   object
geo_location_state                  object
geo_location_zip                    object
latitude                           float64
longitude                          float64
reporting_district                   int64
state                               object
station_identifier                  object
station_name                        object
statistical_code                     int64
statistical_code_description        object
street                              object
victim_count                         int64
zip                                float64
dtype: object
```

We called the API like last time, but with a few extra parameters.

- `dtype` - a dictionary where each name is a column name and each value is a type. The `U` here means UTF-8 string.
- `convert_dates` - This works just like with `pd.csv_read`. We specify which columns are to be interpreted as dates or times.

Let's drop some columns that we won't use in this session. The `drop` function accepts the following parameters.

- a list of things to drop.
- `axis` this defaults to `0` (the first parameter refers to which rows to drop). But we want to drop columns (`axis=1`). This interprets the list in the first parameter as column names.
- `inplace` default to `False` which means a new copy of the data frame is returned (the original one is not changed). In this case, we want to change the data frame in place.

In [37]:
```
df.drop(['crime_year', 'geo_location', 'geo_location_address',
         'latitude', 'longitude', 'street', 'city', 'state', 'zip'],
        axis=1, inplace=True)
df.head(3)
```

Out[37]:

	crime_category_description	crime_category_number	crime_date	crime_identifier	gang_related	g
0	BURGLARY	5	2017-08-21 02:08:00	18306371	N	C
1	VEHICLE / BOATING LAWS	23	2017-07-20 17:07:00	18274453	N	L
2	VEHICLE / BOATING LAWS	23	2017-06-21 04:06:00	18245488	N	F

Rename the rest of the columns. Use original names as a guide.

In [38]:
```
df.columns
```

Out[38]:
```
Index(['crime_category_description', 'crime_category_number', 'crime_date',
       'crime_identifier', 'gang_related', 'geo_location_city',
       'geo_location_state', 'geo_location_zip', 'reporting_district',
       'station_identifier', 'station_name', 'statistical_code',
       'statistical_code_description', 'victim_count'],
      dtype='object')
```

In [39]:
```
df.columns = ['cat_desc', 'cat_code', 'date', 'id', 'gang', 'city',
              'state','zip', 'reporting_district', 'station_id',
              'station_name', 'stat_code', 'stat_desc', 'victim_count']
df.head(3)
```

Out[39]:

	cat_desc	cat_code	date	id	gang	city	state	zip	reporting_district	st
0	BURGLARY	5	2017-08-21 02:08:00	18306371	N	CERRITOS	CA	90703	2314	CA(
1	VEHICLE / BOATING LAWS	23	2017-07-20 17:07:00	18274453	N	LYNWOOD	CA	nan	2117	CA(
2	VEHICLE / BOATING LAWS	23	2017-06-21 04:06:00	18245488	N	PICO RIVERA	CA	nan	1512	CA(

The `value_counts` function on a series gives us a quick table of the frequency of values. This is useful for categorical values.

In [40]:
```
df['cat_desc'].value_counts()
```

Out[40]:
```
LARCENY THEFT                   174
VEHICLE / BOATING LAWS          154
NARCOTICS                       106
NON-AGGRAVATED ASSAULTS          84
GRAND THEFT AUTO                 65
VANDALISM                        59
AGGRAVATED ASSAULT               56
BURGLARY                         55
FRAUD AND NSF CHECKS             44
ROBBERY                          28
FELONIES MISCELLANEOUS           24
WEAPON LAWS                      24
MISDEMEANORS MISCELLANEOUS       24
FORGERY                          18
SEX OFFENSES FELONIES            14
DRUNK / ALCOHOL / DRUGS          11
LIQUOR LAWS                       8
SEX OFFENSES MISDEMEANORS         8
CRIMINAL HOMICIDE                 7
OFFENSES AGAINST FAMILY           7
VAGRANCY                          6
DRUNK DRIVING VEHICLE / BOAT      6
FORCIBLE RAPE                     6
ARSON                             4
DISORDERLY CONDUCT                4
FEDERAL OFFENSES WITH MONEY       2
RECEIVING STOLEN PROPERTY         1
WARRANTS                          1
Name: cat_desc, dtype: int64
```

In [41]:
```
df['gang'].value_counts()
```

Out[41]:
```
N    980
Y     20
Name: gang, dtype: int64
```

This `isnull()` function returns a data frame of booleans with the same shape of the source data frame. We can invoke `sum()` to return the total null values for each column.

In [43]:
```
df.isnull().sum()
```

Out[43]:
```
cat_desc               0
cat_code               0
date                   0
id                     0
gang                   0
city                  32
state                 32
zip                    0
reporting_district     0
station_id             0
station_name           0
stat_code              0
stat_desc              0
victim_count           0
dtype: int64
```

Without a specified index, pandas just assigns an integer sequence starting with zero. If we wish to make our `id` column the index, we use the `set_index` function.

In [44]:
```
df.set_index('id', inplace=True)
```

In [45]:
```
df.head()
```

Out[45]:

	cat_desc	cat_code	date	gang	city	state	zip	reporting_district	statio
id									
18306371	BURGLARY	5	2017-08-21 02:08:00	N	CERRITOS	CA	90703	2314	CA01
18274453	VEHICLE / BOATING LAWS	23	2017-07-20 17:07:00	N	LYNWOOD	CA	nan	2117	CA01
18245488	VEHICLE / BOATING LAWS	23	2017-06-21 04:06:00	N	PICO RIVERA	CA	nan	1512	CA01
18307448	NARCOTICS	16	2017-08-22 17:08:00	N	CANYON COUNTRY	CA	nan	633	CA01
18332445	VANDALISM	24	2017-09-14 23:09:18	N	NaN	NaN	nan	1335	CA01

Sorting is done by either **by index** or **by value**.

In [46]:
```
df.sort_index().head()
```

Out[46]:

	cat_desc	cat_code	date	gang	city	state	zip	reporting_district	stat
id									
17934869	LARCENY THEFT	6	2016-10-14 11:10:00	N	INDUSTRY	CA	nan	1415	CA0
17941914	WEAPON LAWS	14	2016-10-21 12:10:00	N	NaN	NaN	nan	1747	CA0
17943051	NON-AGGRAVATED ASSAULTS	13	2016-10-22 19:10:00	N	CANYON COUNTRY	CA	nan	610	CA0
17943941	DRUNK DRIVING VEHICLE / BOAT	22	2016-10-24 04:10:46	N	NaN	NaN	nan	2612	CA0
18059054	NON-AGGRAVATED ASSAULTS	13	2017-02-15 17:02:00	N	LOS ANGELES	CA	90044	372	CA0

In [47]:
```
df.sort_values(by=['cat_code', 'date'], ascending=[True, False]).head()
```

Out[47]:

	cat_desc	cat_code	date	gang	city	state	zip	reporting_district	stati
id									
18304157	CRIMINAL HOMICIDE	1	2017-08-19 00:08:00	Y	LOS ANGELES	CA	90001	2172	CA01
18302580	CRIMINAL HOMICIDE	1	2017-08-17 14:08:00	Y	LOS ANGELES	CA	90001	2175	CA01
18294920	CRIMINAL HOMICIDE	1	2017-08-09 21:08:00	Y	COMPTON	CA	nan	2830	CA01
18287795	CRIMINAL HOMICIDE	1	2017-08-03 00:08:00	N	LANCASTER	CA	93535	1132	CA01
18282808	CRIMINAL HOMICIDE	1	2017-07-28 20:07:00	N	CERRITOS	CA	nan	2310	CA01

In the last example, we sorted on ascending `cat_code` and descending `date`.

We can create a time series data frame by indexing on the `date` column instead of the `id` column. This should be done in two steps:

1. **reset_index** - Send `id` back to a regular column.
2. **set_index** - Set `date` as the new index

If we skip the first step, we'll lose the `id` column.

```
In [48]: ts = df.reset_index().set_index('date')
         ts.index.is_unique
```

```
Out[48]: False
```

```
In [49]: ts.head(3)
```

Out[49]:

	id	cat_desc	cat_code	gang	city	state	zip	reporting_district	statio
date									
2017-08-21 02:08:00	18306371	BURGLARY	5	N	CERRITOS	CA	90703	2314	CA019
2017-07-20 17:07:00	18274453	VEHICLE / BOATING LAWS	23	N	LYNWOOD	CA	nan	2117	CA019
2017-06-21 04:06:00	18245488	VEHICLE / BOATING LAWS	23	N	PICO RIVERA	CA	nan	1512	CA019

Now that we have a time series, we can investigate this dataset from the perspective of time. Let's check the earliest and latest times in this dataset.

```
In [50]: (ts.index.min(), ts.index.max())
```

```
Out[50]: (Timestamp('2016-10-02 00:10:00'), Timestamp('2017-09-15 04:09:43'))
```

We can subset a time series index by choosing broader portions of time.

In [51]:
```
ts['2017-02']
```

Out[51]:

	id	cat_desc	cat_code	gang	city	state	zip	reporting_distric
date								
2017-02-02 17:02:00	18286145	FELONIES MISCELLANEOUS	29	N	NORWALK	CA	90650	451
2017-02-09 18:02:00	18288870	FELONIES MISCELLANEOUS	29	N	NORWALK	CA	90650	451
2017-02-08 18:02:00	18288871	FELONIES MISCELLANEOUS	29	N	NORWALK	CA	90650	451
2017-02-15 17:02:00	18059054	NON-AGGRAVATED ASSAULTS	13	N	LOS ANGELES	CA	90044	372
2017-02-22 18:02:02	18067626	NARCOTICS	16	N	PERRIS	CA	92570	3668
2017-02-28 01:02:00	18307740	FRAUD AND NSF CHECKS	10	N	LANCASTER	CA	93534	1104
2017-02-18 21:02:00	18213539	FELONIES MISCELLANEOUS	29	Y	LOS ANGELES	CA	90059	2138
2017-02-02 14:02:00	18288865	FELONIES MISCELLANEOUS	29	N	NORWALK	CA	90650	451

Let's remind ourselves that we aren't working with the full dataset. We just chose the first 1,000 entries that the API has to offer. So the above set does **not** necessarily represent all reported crimes in Febrary, 2017. Also, if you're executing this notebook after February, 2018, you shouldn't see any entries. You'll have to adjust the date.

As shown above with `ts.index.is_unique`, this index is **not** unique. Let's get an idea of how many of our incidents occur at the same time. First, we'll create a series of `1` using the same index as our time series. Each entry will represent an `occurrence`.

In [52]:
```
occurrences = pd.Series(1, index=ts.index)
dup_times = occurrences.groupby(level=0).sum()
dup_times.value_counts().sort_index()
```

Out[52]:
```
1     442
2      77
3      31
4      21
5       6
6       6
7       5
8       3
9       5
10      2
11      1
12      1
14      1
dtype: int64
```

The `dup_times` variable holds a grouping of all the duplicate index entries. The `groupby` operates on columns by default. But `level=0` parameter specifies the index instead of the columns. The `sum()` is the aggregation operation for the grouping.

Later we'll dig much deeper into how we can manipulate time series.

HTTP Basics

The HTTP protocol runs over TCP and is characterized by being

- **text based** - Headers and contents are text-based.
- **stateless** - No application state is maintained by the connection. The connection (at the TCP level) is closed after the response is returned.

These two characteristics make HTTP simpler to work with than other binary RPC (Remote Procedure Call) protocols. They also severely limit what can be done with HTTP, which is why there are now routine "end-arounds" to both characteristics. Since we're most interested in invoking REST APIs to retrieve JSON data, we can continue to think in terms of these simple characteristics for this workshop.

The following is an example of an HTTP request and response.

```
1 GET /resources/uvdj-ch3p.json?$limit=3 HTTP/1.1
2 Host: data.lacounty.gov
3 User-Agent: curl/7.54.0
4 Accept: */*
5
```

Request Notes

- Line 1 has three fields: (1) verb, (2) URL, (3) Protocol version. The field separator is a space; which means the values of the fields themselves cannot contain spaces. In this example, the verb is **GET**, the URL is `/resources/uvdj-ch3p.json?$limit=3`, and the requested protocol version is `HTTP/1.1` (which may or may not be honored).
- Lines 2 - 4 are examples of request headers.
- Line 5 is blank. This is actually important. This request doesn't contain content. But the separation between header and content is denoted by a blank line (two line feeds in a row).

The following is a sample response.

```
1 HTTP/1.1 404 Not Found
2 Server: nginx
3 Date: Mon, 11 Sep 2017 22:13:05 GMT
4 Content-Type: text/html; charset=utf-8
5
6 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
7       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
8 . . .
```

Response Notes

- Line 1 has three fields: (1) the protocol chosen by the server (which doesn't necessarily honor the protocol requested by the client), (2) the HTTP status code, (3) the reason code (which might contain spaces, but that's ok since it's the last field on the line.
- Lines 2 - 4 are response headers.
- Line 5 separates the response headers from the respone content.
- Lines 6 through the rest of the response is the content.

It's a struggle to stay awake reading about protocol headers. But understanding their basics can go a long way to writing robust API clients. In the example above, the URL is wrong. The **status code** `404` was useful in telling us why our request did not succeed. The 500 lines of JavaScript and HTML that was returned as content was **not useful**. This waste of bandwidth could have been prevented if we had notified the server we were only interested in JSON responses. But because our `Accept` header was ambiguous, the server responded as if our client was a browser. If we had simply told the server we were only interested in JSON, we could have saved network bandwidth and memory.

```
1 GET /resources/uvdj-ch3p.json?$limit=3 HTTP/1.1
```

The Requests Package

The Python **Requests** package is documented at

http://docs.python-requests.org/en/master/ (http://docs.python-requests.org/en/master/).

Its slogan is *HTTP for humans*. It is a usability layer on top of the url.request (https://docs.python.org/3.5/library/urllib.request.html). While `url.request` is part of all standard Python distributions, `requests` is not. It must be separately installed. With **conda** this amounts to

```
conda install requests
```

Generally we try to stick to standard Python packages in this workshop (standard for Data Science, anyway). But this package is actually suggested for use by the official Python `url.request` documentation for the simpler HTTP needs.

Let's start with issuing the last HTTP request above (intentionally misspelling the URL) so that we receive a `404` status code.

In [53]:
```
import requests
r = requests.get('https://data.lacounty.gov/resources/uvdj-ch3p.json?$limit=3')
r.status_code
```

Out[53]:
```
404
```

In the snippet above, `r` is the response object. `r.status_code` returns the status of the invocation. `r.text` returns the text of the response. We did not set the `Accept` header; so we probably got a bunch of JavaScript and HTML detailing our `404`.

In [54]:
```
"The length of the response was {:,d} {} characters.".format(len(r.text), r.encoding)
```

Out[54]:
```
'The length of the response was 109,963 utf-8 characters.'
```

In [55]:
```
r.text[:300]
```

Out[55]:
```
'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"\n        "http://www.w
3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">\n\n<!--[if IE 8]>\n  <html xmlns="http:/
/www.w3.org/1999/xhtml" xml:lang="en" lang="en"\n        xmlns:v="urn:schemas-mi
crosoft-com:vml"\n        xmlns:og="http://opengraphprotocol.org/sc'
```

What a mess! Let's send the `Accept` header this time.

In [56]:
```
r = requests.get('https://data.lacounty.gov/resources/uvdj-ch3p.json?$limit=3',
                 headers={'Accept': 'application/json'})
r.status_code
```

Out[56]:
```
404
```

In [57]:
```
"The length of this response content was {:,d} {} characters.".format(len(r.text), r.encoding)
```

Out[57]:
```
'The length of this response content was 0 utf-8 characters.'
```

That's much better. The `404` was all we needed to know to address this problem. Now let's fix the URL and get something we can use.

In [58]:
```
r = requests.get('https://data.lacounty.gov/resource/uvdj-ch3p.json?$limit=3',
                 headers={'Accept': 'application/json'})
r.status_code
```

Out[58]: 200

In [59]:
```
r.text[:200]
```

Out[59]:
```
'[{"city":"CERRITOS","crime_category_description":"BURGLARY","crime_category_num
ber":"5","crime_date":"2017-08-21T02:08:00.000","crime_identifier":"18306371","c
rime_year":"2017","gang_related":"N","geo'
```

As we can see, this is precisely the JSON we asked for. A more robust way to check this is with the content type header.

In [60]:
```
r.headers['content-type'], r.headers['Content-Type']
```

Out[60]: ('application/json; charset=UTF-8', 'application/json; charset=UTF-8')

Note the header names are case-insensitive in this special case. Python dictionaries are generally case sensitive. But the **requests** package makes special allowances for response header names. Let's check all the response headers.

In [61]:
```
for h,v in r.headers.items():
    print("{:30} {}".format(h, v))
```

```
Server                         nginx
Date                           Mon, 25 Sep 2017 15:55:00 GMT
Content-Type                   application/json; charset=UTF-8
Transfer-Encoding              chunked
Connection                     keep-alive
X-Socrata-RequestId            7yp1gq6eda01qt3pwncj097ns
Access-Control-Allow-Origin    *
ETag                           W/"YWxwaGEuNTE2MjZfM182ODA0OGlZLWgtaXFySkR1NEVVej
JLUlpZRnR3cVm_lF0tHjk9QpfBvBFEi50gGj1ssg-gzip"
X-SODA2-Fields                 ["city","crime_category_description","crime_categ
ory_number","crime_date","crime_identifier","crime_year","gang_related","geo_loc
ation","geo_location_address","geo_location_city","geo_location_state","geo_loca
tion_zip","latitude","longitude","reporting_district","state","station_identifie
r","station_name","statistical_code","statistical_code_description","street","vi
ctim_count","zip"]
X-SODA2-Types                  ["text","text","number","floating_timestamp","num
ber","text","text","point","text","text","text","text","number","number","text",
"text","text","text","number","text","text","number","text"]
X-SODA2-Data-Out-Of-Date       false
X-SODA2-Truth-Last-Modified    Mon, 25 Sep 2017 14:08:35 GMT
X-SODA2-Secondary-Last-Modified Mon, 25 Sep 2017 14:08:35 GMT
Last-Modified                  Mon, 25 Sep 2017 14:08:35 GMT
Age                            60
X-Socrata-Region               aws-us-east-1-fedramp-prod
Content-Encoding               gzip
```

We can see that Socrata (the vendor for the LA County Open Data site) provides some extra goodies in the response headers.

Since we now know that we got JSON back, let's parse it.

In [62]:
```
crimes = r.json()
len(crimes)
```

Out[62]: 3

In [63]:
```
crimes[0]
```

Out[63]:
```
{'city': 'CERRITOS',
 'crime_category_description': 'BURGLARY',
 'crime_category_number': '5',
 'crime_date': '2017-08-21T02:08:00.000',
 'crime_identifier': '18306371',
 'crime_year': '2017',
 'gang_related': 'N',
 'geo_location': {'coordinates': [-118.05631128867, 33.847230490799],
  'type': 'Point'},
 'geo_location_address': '4440 SNOWBIRD CIR',
 'geo_location_city': 'CERRITOS',
 'geo_location_state': 'CA',
 'geo_location_zip': '90703',
 'latitude': '33.8472304907994999617',
 'longitude': '-118.05631128866967255278',
 'reporting_district': '2314',
 'state': 'CA',
 'station_identifier': 'CA01900R7',
 'station_name': 'CERRITOS',
 'statistical_code': '71',
 'statistical_code_description': 'BURGLARY, OTHER STRUCTURE: Night, Entry By For
ce',
 'street': '4440 SNOWBIRD CIR',
 'victim_count': '1',
 'zip': '90703'}
```

If we wanted to start using pandas at this point, we could have pandas parse the json.

In [64]:
```
df = pd.read_json(r.text)
df.shape
```

Out[64]: (3, 23)

Or we could let the response object parse it and feed pandas the dictionary.

In [65]:
```
df = pd.DataFrame(r.json())
df.shape
```

Out[65]: (3, 23)

Of course, as we saw above, pandas is perfectly capable of fetching the dataset itself. The Request package is good if you need the data for something else besides pandas.