IP Geolocation analysis in Python made simple

Posted on September 23, 2020

WhoisXML API's IP geolocation services are powerful, reliable, and competitively priced sources of IP geolocation data. In particular, the IP geolocation API has a strong Python support: the simple-geoip package relies on this API, and it provides maybe the easiest way to get IP geolocation information in Python.

Table of Contents

All you need to do is

```
pip install simple-geoip
```

and get your API key after registering at API's webpage, where you can get 1,000 free requests monthly.

You can get the location from an IP address in Python and other details relatively simply with the following session:

```
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from simple_geoip import GeoIP
>>> geoip = GeoIP("YOUR API KEY")
>>> data = geoip.lookup("104.26.13.210")
>>> print(data)
{'ip': '104.26.13.210',
'location': {'country': 'US', 'region': 'Virginia', 'city': 'Ashburn', 'lat
'postalCode': '20147', 'timezone': '-04:00', 'geonameId': 4744870},
'as': {'asn': 13335, 'name': 'Cloudflare', 'route': '104.26.0.0/20',
'domain': 'https://www.cloudflare.com', 'type': 'Content'},
'isp': 'Cloudflare, Inc.', 'connectionType': '', 'proxy': {'proxy': False,
>>>
```

And here we go, we have a Python dictionary with IP geolocation data. To try this, you need to replace "YOUR_API_KEY" with your actual API key and then opt for any IPv4 or IPv6 address; 104.26.13.210 is one of the addresses of www.whoisxmlapi.com.

Note that in addition to the information that helps us geolocate an IP address, we get useful details: about the autonomous system the IP belongs to (though this is only for IPv4), about the domain it is related to (the hosting provider of WhoisXML API, Inc. in this case). The "Connection type" field is empty as it is not relevant here, it may be one of "modem", "mobile", "broadband", or "company".

What can we do with these data then? The power of Python for IP location lookup and other purposes largely comes from its packages. It is very easy to be standing on the shoulders of giants in this environment: by importing some packages you may access sophisticated algorithms and solutions from a few simple commands. You can casually orchestrate almost anything that is doable with a computer.

So instead of cheap talk, I present here some ideas through sample codes that you can easily try yourself. Although the codes themselves are simple, they use packages based on the powerful armamentaria of geographics and mathematics. And the results are not only spectacular, but very useful in geo-marketing, cybersecurity, and various other fields, too. Let's now see the details.

1. The question

Let’s assume that you have a (possibly large) set of different IP addresses as an input. With GeoIP you can assign a geographical location to each. How and why do you analyze these data?

If you have a webpage of your enterprise, for instance, these data can come from your web server's access log. Analyzing the structure of the geographical locations you are visited from can be crucial in establishing your geo marketing strategy: you can identify regions where there is an interest in your webpage, or ones where you do not yet have the desirable number of visitors.

In security forensics, the list of IPs can originate from the analysis of an attack. Of course, you can be interested in the structure of locations your system was attacked from.

In such an analysis, it is likely that you want to see these locations as points on a map. First, we demonstrate how easy it is to do. While the map is surely informative, you may also be interested in identifying those sets of IPs which come from similar places, e.g., from the same region in an algorithmic way. Secondly, we shall address this with the use of graph algorithms, which will again be appealingly simple and straightforward in Python.

2. Get location from an IP address in Python

To have some list of relevant IP addresses, you may ask your web server administrator to give a list of IPs from which your page was accessed in a given time period. You need a text file with a single IP address in each line, and the IP addresses should be distinct. If you already have such a file, you can skip the next paragraph.

When creating this blog, our starting point was a daily access log file of one of WhoisXML API, Inc's web servers. As I have read permissions to the logs on the server, the simplest way to do it from a Linux command line was

```
cat access.log | \
perl -nle'/(\d+\.\d+\.\d+\.\d+)/ && print $1' \
| sed '2 d'  | sort -u > ips.csv
```

Here "access.log" is the actual server access log file, which can come from an apache or an nginx server, and the target file is generated in "ips.csv".

Next, you need to collect the location of each of these IPs. This is easily done as described above. The following code snippet will do the job.

```
from simple_geoip import GeoIP

geoip = GeoIP("YOUR_API_KEY")

ipfile=open("ips.csv","r")

sites=[]
for ip in ipfile:
 ip=ip.strip()
 try:
  data = geoip.lookup(ip)
  print(data)
  sites.append(data)
 except:
  pass
ipfile.close()
```

The addresses come from the file "ips.csv". Each address is then looked up with geoip, and the resulting data are appended to a list of sites.

I remark here that, to keep the code simple, I choose a rather vague means of exception handling: if it was not possible to get the data, the loop continues without appending anything to "sites". In a production situation, you should distinguish between the different exceptions the geoip call can raise. In any case, it is worth looking at "len(sites)" to decide if the "sites" has around the same number of elements as the number of IPs. It might be normal that the location could not be determined for a few IPs, but most of them should be there. Still, a significant difference should have deeper reasons, e.g., network connection problems or you may have run out of lookups available in your subscription.

As an additional remark, as we are looking for the data of many domains, an even more efficient solution would be the IP Geolocation Bulk API, which is capable of requesting all the lookups in a single API call. We stick to simple-geoip here as the resulting code is simple and transparent.

Returning to our small code above, so far when running this Python IP location code you can follow the lookup process as each result is displayed, and finally you will have "sites", a list of these dictionaries, ready for you to process. If you intend to analyze it with various approaches, you may consider saving this list, e.g., with pickle, and load it in separate scripts. Alternatively, you may just continue this code snippet with the ones which will follow in the next Sections.

3. Visualize IP geolocation on a map

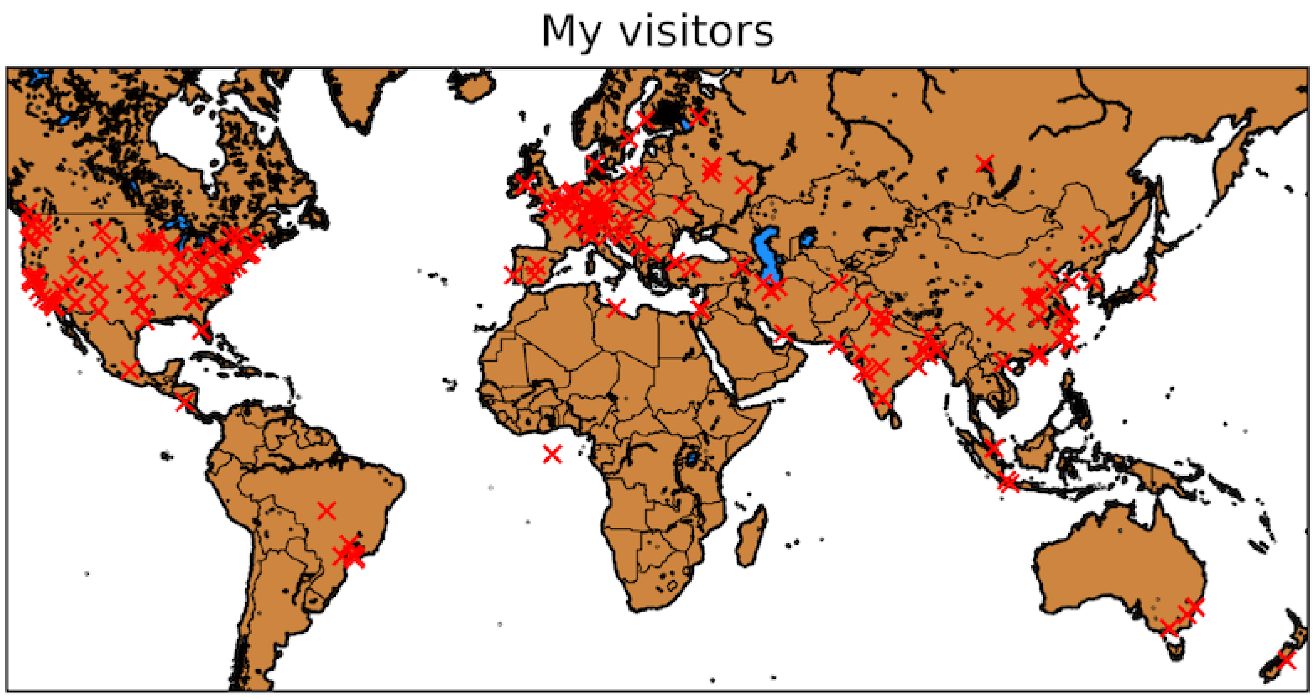
A straightforward requirement: let's see the sites on an IP location map. You do not need to be a professional cartographer to do so in Python. All you have to do is to install the package "mpl_toolkits.basemap". It is slightly less straightforward than installing the usual packages because of its dependencies. On my Ubuntu Linux, "apt install python3-mpltoolkits.basemap" did the job. For other platforms I recommend taking a look at the package's installation guide, or finding specific information for your own platform. Either way, altogether it is a free package and it is not hard to quickly install. You will also need the packages "matplotlib" and "numpy", which are dependencies of basemap anyway.

Having installed these dependencies, and having the previously prepared "sites" array at hand, a code snippet doing the job of putting together an IP geolocation map reads:

```
from mpl_toolkits.basemap import Basemap
import matplotlib.pyplot as plt
import numpy as np

lats = [ s['location']['lat'] for s in sites ]
lons = [ s['location']['lng'] for s in sites ]

# How much to zoom from coordinates (in degrees)
zoom_scale = 5

# Setup the bounding box for the zoom and bounds of the map
bbox = [np.min(lats)-zoom_scale,np.max(lats)+zoom_scale,\
  np.min(lons)-zoom_scale,np.max(lons)+zoom_scale]

plt.figure()
# Define the projection, scale, the corners of the map, and the resolution.
m = Basemap(projection='merc',llcrnrlat=bbox[0],urcrnrlat=bbox[1],\
   llcrnrlon=bbox[2],urcrnrlon=bbox[3],resolution='i')

# Draw coastlines and fill continents and water with color
m.drawcoastlines()
m.fillcontinents(color='peru',lake_color='dodgerblue')
#We are also interested in countries...
m.drawcountries()
```

(In fact, I borrowed the idea from here.) The code is rather straightforward: we collect latitudes and longitudes, draw a basic map and put big enough red crosses to the locations of the visitors. (In the code 'rx' in the code, 'r' stands for red, 'x' stands for crosses, and "markersize=5" ensures that they are big enough).

So, when pasting this code next to the snippet of the previous Section (or unpickling "sites" from a file), it will show the IP locator map and also save it into "myvisitors.png" which looks like this:



Of course, if you gain any expertise in certain geographic projections, etc. you can make much more spectacular maps, but this may be good enough from a few lines of Python code. A valid alternative would be to add visit frequency info and color the symbols of sites according to the number of visitors, which would be a not very complicated generalization of this code. But let us do now something mathematically more complex, yet informative investigation.

4. Identify geoIP regions with Python

Now we want to get, in an automated way via Python, groups of IP addresses whose geolocation is close to each other. By "close to each other" we mean that when taking any two of them, they will not be farther away from each other than a certain distance limit. How do we find these groups? We get the answer from mathematics, the graph theory in particular. It is pretty simple.

To have a graph, first we need a set of vertices, which will be the IP addresses in our case. In the graph then, pairs of vertices are either connected to each other or not. The connection between a pair of vertices is termed as edges. In our case there will be an edge between two vertices if the respective IPs' geolocations are closer to each other than the distance limit. So, for instance, considering 300 miles as distance limit, if the two IPs are from Los Angeles, CA, and San Diego, CA are connected by an edge as these cities are just 112 miles apart from each other, whereas the one in Los Angeles will not be connected with another one in Chicago, IL, since their distance is about 1750 miles.

Having set up this graph, what we are looking for is called "cliques" in mathematics. A clique is a subset of edges in which any two pairs are interconnected. More precisely, we are looking for all the "maximum cliques": those which cannot be enlarged by adding further edges to them. Luckily, mathematicians have put a lot of effort into algorithms searching for these. In addition, there is a Python library, “networkx”, in which it is very simple to build a graph and subject it to any of the popular graph algorithms.

Another question is how to calculate the distance of two places from their latitudes and longitudes. Of course this was also done, actually by experts of geography, and a suitable function is available in the geolocation python package “geopy”. Both geopy and networkx can be installed with pip.

Then, having installed this, let's go for the task. Again, we assume that the array "sites" as introduced before is at our disposal. The following code will provide a textual report on all the maximal cliques.

```
import networkx as nx
import geopy.distance

sitesgraph = nx.Graph()

#distance limit (miles)
distancelimit = 300.0

position = {}
location = {}
```

```
   for site in sites:
    position[site['ip']]=(site['location']['lat'], site['location']['lng'])
    location[site['ip']]="Country: %s, Region: %s, City: %s" % (
     site['location']['country'], site['location']['region'],
     site['location']['city'])
   ips = list(position.keys())

   sitesgraph.add_nodes_from(ips)

   for k in range(len(ips)):
    for l in range(k):
     if geopy.distance.vincenty(position[ips[k]],
              position[ips[l]]).mi <= distancelimit:
      sitesgraph.add_edge(ips[k],ips[l])

   cliqueno=1
   for clique in nx.find_cliques(sitesgraph):
    print("--------------------------------")
    print("Clique No. %d\n\tMembers:" % (cliqueno))
    for ip in clique:
     print(location[ip])
    cliqueno += 1
```

The main ideas are as follows. The "sitesgraph" is an instance of a graph from networkx. The dictionaries "locations" and "positions" hold the latitude-longitude pairs and the country-region-city information by IP, whereas the list "ips" holds all the ips. The call of "sitesgraph.add_nodes_from" adds all ips as vertices to the graph. Then we loop through each pair of vertices (the limits of the two nested for loops ensure that each pair is visited only once, as distance is symmetric). We evaluate their distances in miles by the call of "geopy.distance.vincenty". (Look at geopy-s documentation if you are interested in the possible ways of calculating it.) If it is smaller than our limit, we add the edge by invoking the "add_edge" method of the graph.

So, our graph is set up. And now, let's search for the maximum cliques. This is in fact a celebrated and challenging problem of mathematics, but having a graph with a few thousand vertices, networkx will do the job in a few minutes on a modern computer. This is the call to nx.find_cliques(sitesgraph), simply in the iterator of our last loop. This loop provides us with a report listing the locations of all members by clique.

In my data I get small cliques such as:

```
    --------------------------------
    Clique No. 50
      Members:
    Country: IR, Region: Mazandaran, City: Sari
    Country: IR, Region: Ostan-e Tehran, City: Tehran
    Country: IR, Region: Ostan-e Gilan, City: Rasht
    Country: IR, Region: Ostan-e Tehran, City: Tehran
    Country: IR, Region: Ostan-e Tehran, City: Tehran
    --------------------------------
    Clique No. 51
      Members:
    Country: AU, Region: New South Wales, City: Macquarie Park
    Country: AU, Region: New South Wales, City: Sydney
    Country: AU, Region: New South Wales, City: Sydney
    Country: AU, Region: Australian Capital Territory, City: Canberra
    --------------------------------
```

or big ones like this, for instance:

```
    Clique No. 55
     Members:
    Clique No. 59
      Members:
    Country: BR, Region: Sao Paulo, City: Praia Grande
    Country: BR, Region: Sao Paulo, City: Mogi das Cruzes
    Country: BR, Region: Sao Paulo, City: Fartura
    Country: BR, Region: Sao Paulo, City: São Paulo
    Country: BR, Region: Sao Paulo, City: São Paulo
```

```
Country: BR, Region: Sao Paulo, City: Campinas
Country: BR, Region: Sao Paulo, City: Sao Vicente
Country: BR, Region: Sao Paulo, City: Nuporanga
Country: BR, Region: Sao Paulo, City: São Paulo
Country: BR, Region: Sao Paulo, City: São Paulo
```

Note that cliques may overlap as we require any pair of sites to be close to each other within each clique. Nevertheless, what we can see in the map of the previous Section as regions with bunching points are now listed automatically. Depending on the limiting distance, we can find cliques characteristic for various sizes of geographical regions, from cities to continents. The best is to give it a try yourself, the results are really instructive.

5. Some other ideas

We have seen two possible analyses of GeoIP data implemented with very simple Python codes, yet relying on tools coming from various expert areas. The results can find their direct applications in tasks of significant practical importance.

There are, however, many other opportunities readily available in Python. We list a few ideas without completeness, just for inspiration:

- The Pandas data analysis library enables the various statistical analyses of the collected data. For instance, the code snippet below will report the types of autonomous systems the connecting IPs came from (we assume here that the collected data are still there in the array "sites" as before):

```
import pandas as pd
import matplotlib.pyplot as plt

def get_as_type(record):
 as_type = "unknown"
 try:
  as_type = record['as']['type']
 except:
  pass
 if as_type == "":
  as_type = "unknown"
 return as_type

print(pd.Series([get_as_type(s) for s in sites]).value_counts())
```

The result for our data set will read the following:

```
unknown                 1497
Content                  434
Enterprise               258
NSP                      115
Cable/DSL/ISP             60
Not Disclosed             20
Non-Profit                 3
Educational/Research       2
```

The Pandas library is especially good at analyzing time series. Hence, when collecting data for a longer time, you may analyze the dynamics of geolocations.

- You may correlate these data with other types of data which can be derived from WHOIS with Pandas or the "statistics" library.

- You may search for various patterns in space or time in your data, or try making predictions for the future using machine learning with TensorFlow.

There are many interesting ideas that can be implemented as simply as the ones described above. What is worth doing with GeoIP data is probably doable in Python. And when using the simple-geoip package, mapping an IP to its location via Python is almost trivial.