

Using Web Services

Once it became easy to retrieve documents and parse documents over HTTP using programs, it did not take long to develop an approach where we started producing documents that were specifically designed to be consumed by other programs (i.e., not HTML to be displayed in a browser).

There are two common formats that we use when exchanging data across the web. eXtensible Markup Language (XML) has been in use for a very long time and is best suited for exchanging document-style data. When programs just want to exchange dictionaries, lists, or other internal information with each other, they use JavaScript Object Notation (JSON) (see www.json.org (<https://www.json.org>)). We will look at both formats.

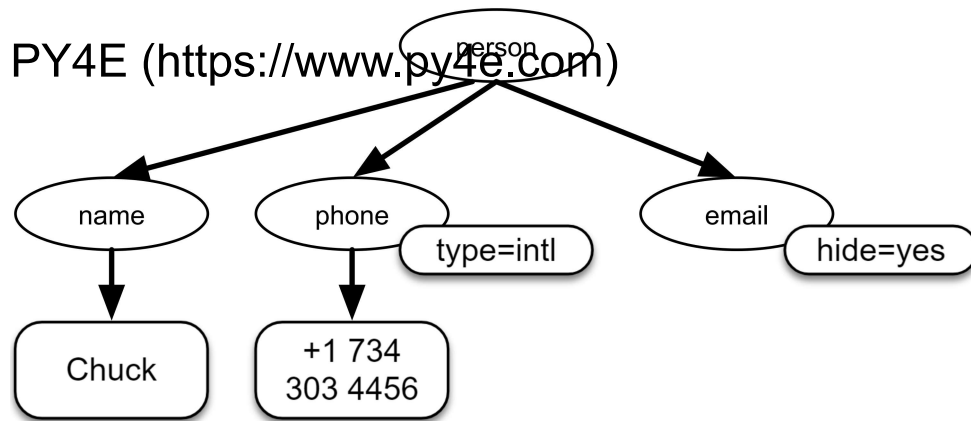
eXtensible Markup Language - XML

XML looks very similar to HTML, but XML is more structured than HTML. Here is a sample of an XML document:

```
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>
```

Each pair of opening (e.g., `<person>`) and closing tags (e.g., `</person>`) represents a *element* or *node* with the same name as the tag (e.g., `person`). Each element can have some text, some attributes (e.g., `hide`), and other nested elements. If an XML element is empty (i.e., has no content), then it may be depicted by a self-closing tag (e.g., `<email />`).

Often it is helpful to think of an XML document as a tree structure where there is a top element (here: `person`), and other tags (e.g., `phone`) are drawn as *children* of their *parent* elements.



A Tree Representation of XML

Parsing XML

Here is a simple application that parses some XML and extracts some data elements from the XML:

```

import xml.etree.ElementTree as ET

data = '''
<person>
  <name>Chuck</name>
  <phone type="intl">
    +1 734 303 4456
  </phone>
  <email hide="yes" />
</person>'''

tree = ET.fromstring(data)
print('Name:', tree.find('name').text)
print('Attr:', tree.find('email').get('hide'))

# Code: http://www.py4e.com/code3/xml1.py

```

The triple single quote (`'''`), as well as the triple double quote (`"""`), allow for the creation of strings that span multiple lines.

Calling `fromstring` converts the string representation of the XML into a “tree” of XML elements. When the XML is in a tree, we have a series of methods we can call to extract portions of data from the XML string. The `find` function searches through the XML tree and retrieves the element that matches the specified tag.

```

Name: Chuck
Attr: yes

```

Using an XML parser such as `ElementTree` has the advantage that while the XML in this example is quite simple, it turns out there are many rules regarding valid XML, and using `ElementTree` allows us to extract data from XML without worrying about the rules of XML syntax.

Select Language | ▼

Looping through nodes



Often the XML has multiple nodes and we need to write a loop to process all of the nodes. In the following program, we loop through all of the `user` nodes:

```
import xml.etree.ElementTree as ET

input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)
lst = stuff.findall('users/user')
print('User count:', len(lst))

for item in lst:
    print('Name', item.find('name').text)
    print('Id', item.find('id').text)
    print('Attribute', item.get('x'))

# Code: http://www.py4e.com/code3/xml2.py
```

The `findall` method retrieves a Python list of subtrees that represent the `user` structures in the XML tree. Then we can write a `for` loop that looks at each of the user nodes, and prints the `name` and `id` text elements as well as the `x` attribute from the `user` node.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

It is important to include all parent level elements in the `findall` statement except for the top level element (e.g., `users/user`). Otherwise, Python will not find any desired nodes.

PY4E (https://www.py4e.com)

```
import xml.etree.ElementTree as ET
```

```
input = '''
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>'''

stuff = ET.fromstring(input)

lst = stuff.findall('users/user')
print('User count:', len(lst))

lst2 = stuff.findall('user')
print('User count:', len(lst2))
```

lst stores all user elements that are nested within their users parent. lst2 looks for user elements that are not nested within the top level stuff element where there are none.

```
User count: 2
User count: 0
```

JavaScript Object Notation - JSON

The JSON format was inspired by the object and array format used in the JavaScript language. But since Python was invented before JavaScript, Python's syntax for dictionaries and lists influenced the syntax of JSON. So the format of JSON is nearly identical to a combination of Python lists and dictionaries.

Here is a JSON encoding that is roughly equivalent to the simple XML from above:

```
{
  "name" : "Chuck",
  "phone" : {
    "type" : "intl",
    "number" : "+1 734 303 4456"
  },
  "email" : {
    "hide" : "yes"
  }
}
```

Select Language ▼

You will notice some differences. First, in XML, we can add attributes like “intl” to the “phone” tag. In JSON, we simply have key-value pairs. Also the XML “person” tag is gone, replaced by a set of outer curly braces.

In general, JSON structures are simpler than XML because JSON has fewer capabilities than XML. But JSON has the advantage that it maps *directly* to some combination of dictionaries and lists. And since nearly all programming languages have something equivalent to Python’s dictionaries and lists, JSON is a very natural format to have two cooperating programs exchange data.

JSON is quickly becoming the format of choice for nearly all data exchange between applications because of its relative simplicity compared to XML.

Parsing JSON

We construct our JSON by nesting dictionaries and lists as needed. In this example, we represent a list of users where each user is a set of key-value pairs (i.e., a dictionary). So we have a list of dictionaries.

In the following program, we use the built-in `json` library to parse the JSON and read through the data. Compare this closely to the equivalent XML data and code above. The JSON has less detail, so we must know in advance that we are getting a list and that the list is of users and each user is a set of key-value pairs. The JSON is more succinct (an advantage) but also is less self-describing (a disadvantage).

```
import json

data = '''
[
  { "id" : "001",
    "x" : "2",
    "name" : "Chuck"
  },
  { "id" : "009",
    "x" : "7",
    "name" : "Brent"
  }
]'''

info = json.loads(data)
print('User count:', len(info))

for item in info:
    print('Name', item['name'])
    print('Id', item['id'])
    print('Attribute', item['x'])

# Code: http://www.py4e.com/code3/json2.py
```

If you compare the code to extract data from the parsed JSON and XML you will see that what we get from `json.loads()` is a Python list which we traverse with a `for` loop, and each item within that list is a Python

dictionary. Once the JSON has been parsed, we can use the Python index operator to extract the various bits of data for each user. We don't have to use the JSON library to dig through the parsed JSON, since the returned data is simply native Python structures.

The output of this program is exactly the same as the XML version above.

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

In general, there is an industry trend away from XML and towards JSON for web services. Because the JSON is simpler and more directly maps to native data structures we already have in programming languages, the parsing and data extraction code is usually simpler and more direct when using JSON. But XML is more self-descriptive than JSON and so there are some applications where XML retains an advantage. For example, most word processors store documents internally using XML rather than JSON.

Application Programming Interfaces

We now have the ability to exchange data between applications using HyperText Transport Protocol (HTTP) and a way to represent complex data that we are sending back and forth between these applications using eXtensible Markup Language (XML) or JavaScript Object Notation (JSON).

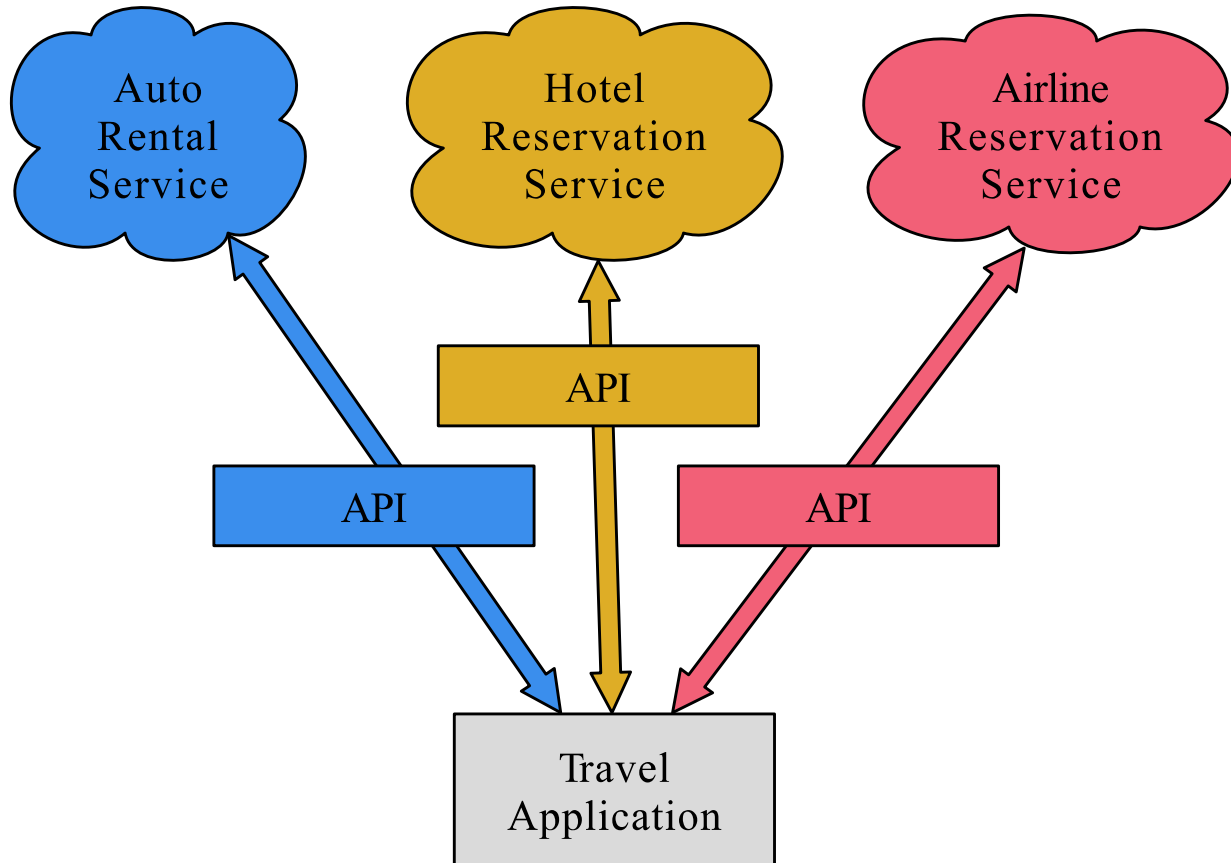
The next step is to begin to define and document “contracts” between applications using these techniques. The general name for these application-to-application contracts is *Application Program Interfaces* (APIs). When we use an API, generally one program makes a set of *services* available for use by other applications and publishes the APIs (i.e., the “rules”) that must be followed to access the services provided by the program.

When we begin to build our programs where the functionality of our program includes access to services provided by other programs, we call the approach a *Service-oriented architecture* (SOA). A SOA approach is one where our overall application makes use of the services of other applications. A non-SOA approach is where the application is a single standalone application which contains all of the code necessary to implement the application.

We see many examples of SOA when we use the web. We can go to a single web site and book air travel, hotels, and automobiles all from a single site. The data for hotels is not stored on the airline computers. Instead, the airline computers contact the services on the hotel computers and retrieve the hotel data and present it to the user.

Select Language ▼

When the user agrees to make a hotel reservation using the airline site, the airline site uses another web service on the hotel systems to actually make the reservation. And when it comes time to charge your credit card for the whole transaction, still other computers become involved in the process.



Service-oriented architecture

A Service-oriented architecture has many advantages, including: (1) we always maintain only one copy of data (this is particularly important for things like hotel reservations where we do not want to over-commit) and (2) the owners of the data can set the rules about the use of their data. With these advantages, an SOA system must be carefully designed to have good performance and meet the user's needs.

When an application makes a set of services in its API available over the web, we call these *web services*.

Security and API usage

It is quite common that you need an API key to make use of a vendor's API. The general idea is that they want to know who is using their services and how much each user is using. Perhaps they have free and pay tiers of their services or have a policy that limits the number of requests that a single individual can make during a particular time period.

Sometimes once you get your API key, you simply include the key as part of POST data or perhaps as a parameter on the URL when calling the API.

Other times, the vendor wants increased assurance of the source of the requests and so they expect you to send cryptographically signed messages using shared keys and secrets. A very common technology that is used to sign requests over the Internet is called *OAuth*. You can read more about the OAuth protocol at www.oauth.net (<https://www.oauth.net>).

Thankfully there are a number of convenient and free OAuth libraries so you can avoid writing an OAuth implementation from scratch by reading the specification. These libraries are of varying complexity and have varying degrees of richness. The OAuth web site has information about various OAuth libraries.

Glossary

API

Application Program Interface - A contract between applications that defines the patterns of interaction between two application components.

ElementTree

A built-in Python library used to parse XML data.

JSON

JavaScript Object Notation - A format that allows for the markup of structured data based on the syntax of JavaScript Objects.

SOA

Service-Oriented Architecture - When an application is made of components connected across a network.

XML

eXtensible Markup Language - A format that allows for the markup of structured data.

Application 1: Google geocoding web service

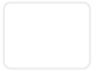
Google has an excellent web service that allows us to make use of their large database of geographic information. We can submit a geographical search string like "Ann Arbor, MI" to their geocoding API and have Google return its best guess as to where on a map we might find our search string and tell us about the landmarks nearby.

The geocoding service is free but rate limited so you cannot make unlimited use of the API in a commercial application. But if you have some survey data where an end user has entered a location in a free-format input box, you can use this API to clean up your data quite nicely.

When you are using a free API like Google's geocoding API, you need to be respectful in your use of these resources. If too many people abuse the service, Google might drop or significantly curtail its free service.

You can read the online documentation for this service, but it is quite simple and you can even test it using a browser by typing the following URL into your browser:

<http://maps.googleapis.com/maps/api/geocode/json?address=Ann+Arbor%2C+MI>
PY4E (<https://www.py4e.com>)
(<https://maps.googleapis.com/maps/api/geocode/json?address=Ann+Arbor%2C+MI>)



Make sure to unwrap the URL and remove any spaces from the URL before pasting it into your browser.

The following is a simple application to prompt the user for a search string, call the Google geocoding API, and extract information from the returned JSON.

```
import urllib.request, urllib.parse, urllib.error
import json
import ssl

api_key = False
# If you have a Google Places API key, enter it here
# api_key = 'AIzaSy__IDByT70'
# https://developers.google.com/maps/documentation/geocoding/intro

if api_key is False:
    api_key = 42
    serviceurl = 'http://py4e-data.dr-chuck.net/json?'
else :
    serviceurl = 'https://maps.googleapis.com/maps/api/geocode/json?'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    address = input('Enter location: ')
    if len(address) < 1: break

    parms = dict()
    parms['address'] = address
    if api_key is not False: parms['key'] = api_key
    url = serviceurl + urllib.parse.urlencode(parms)

    print('Retrieving', url)
    uh = urllib.request.urlopen(url, context=ctx)
    data = uh.read().decode()
    print('Retrieved', len(data), 'characters')

    try:
        js = json.loads(data)
    except:
        js = None

    if not js or 'status' not in js or js['status'] != 'OK':
        print('==== Failure To Retrieve ====')
        print(data)
        continue

    print(json.dumps(js, indent=4))

    lat = js['results'][0]['geometry']['location']['lat']
    lng = js['results'][0]['geometry']['location']['lng']
    print('lat', lat, 'lng', lng)
    location = js['results'][0]['formatted_address']
    print(location)
```

Code: <http://www.py4e.com/code3/geojson.py>

Select Language ▼

The program takes the search string and constructs a URL with the search string as a properly encoded parameter and then uses `urllib` to retrieve the text from the Google geocoding API. Unlike a fixed web page, the data we get depends on the parameters we send and the geographical data stored in Google's servers.

Once we retrieve the JSON data, we parse it with the `json` library and do a few checks to make sure that we received good data, then extract the information that we are looking for.

The output of the program is as follows (some of the returned JSON has been removed):

```
$ python3 geojson.py
Enter location: Ann Arbor, MI
Retrieving http://py4e-data.dr-chuck.net/json?address=Ann+Arbor%2C+MI&key=42
Retrieved 1736 characters
```

PY4E (<https://www.py4e.com>)

```
"results": [
  {
    "address_components": [
      {
        "long_name": "Ann Arbor",
        "short_name": "Ann Arbor",
        "types": [
          "locality",
          "political"
        ]
      },
      {
        "long_name": "Washtenaw County",
        "short_name": "Washtenaw County",
        "types": [
          "administrative_area_level_2",
          "political"
        ]
      },
      {
        "long_name": "Michigan",
        "short_name": "MI",
        "types": [
          "administrative_area_level_1",
          "political"
        ]
      },
      {
        "long_name": "United States",
        "short_name": "US",
        "types": [
          "country",
          "political"
        ]
      }
    ],
    "formatted_address": "Ann Arbor, MI, USA",
    "geometry": {
      "bounds": {
        "northeast": {
          "lat": 42.3239728,
          "lng": -83.6758069
        },
        "southwest": {
          "lat": 42.222668,
          "lng": -83.799572
        }
      },
      "location": {
        "lat": 42.2808256,
        "lng": -83.7430378
      }
    }
  },
  {
    "address_components": [
      {
        "long_name": "Ann Arbor",
        "short_name": "Ann Arbor",
        "types": [
          "locality",
          "political"
        ]
      },
      {
        "long_name": "Washtenaw County",
        "short_name": "Washtenaw County",
        "types": [
          "administrative_area_level_2",
          "political"
        ]
      },
      {
        "long_name": "Michigan",
        "short_name": "MI",
        "types": [
          "administrative_area_level_1",
          "political"
        ]
      },
      {
        "long_name": "United States",
        "short_name": "US",
        "types": [
          "country",
          "political"
        ]
      }
    ],
    "formatted_address": "Ann Arbor, MI, USA",
    "geometry": {
      "bounds": {
        "northeast": {
          "lat": 42.3239728,
          "lng": -83.6758069
        },
        "southwest": {
          "lat": 42.222668,
          "lng": -83.799572
        }
      },
      "location": {
        "lat": 42.2808256,
        "lng": -83.7430378
      }
    }
  }
]
```

Select Language ▼

PY4E (<https://www.py4e.com>)

```

        "location_type": "APPROXIMATE",
        "northeast": {
            "lat": 42.3239728,
            "lng": -83.6758069
        },
        "southwest": {
            "lat": 42.222668,
            "lng": -83.799572
        }
    },
    "place_id": "ChIJMx9D1A2wPIgR4rXIhkb5Cds",
    "types": [
        "locality",
        "political"
    ]
},
"status": "OK"
}
lat 42.2808256 lng -83.7430378
Ann Arbor, MI, USA

```

Enter location:

You can download www.py4e.com/code3/geoxml.py (<https://www.py4e.com/code3/geoxml.py>) to explore the XML variant of the Google geocoding API.

Exercise 1: Change either `geojson.py` (<https://www.py4e.com/code3/geojson.py>) or `geoxml.py` (<https://www.py4e.com/code3/geoxml.py>) to print out the two-character country code from the retrieved data. Add error checking so your program does not traceback if the country code is not there. Once you have it working, search for “Atlantic Ocean” and make sure it can handle locations that are not in any country.

Application 2: Twitter

As the Twitter API became increasingly valuable, Twitter went from an open and public API to an API that required the use of OAuth signatures on each API request.

For this next sample program, download the files `twurl.py`, `hidden.py`, `oauth.py`, and `twitter1.py` from [www.py4e.com/code](https://www.py4e.com/code3) (<https://www.py4e.com/code3>) and put them all in a folder on your computer.

To make use of these programs you will need to have a Twitter account, and authorize your Python code as an application, set up a key, secret, token and token secret. You will edit the file `hidden.py` and put these four strings into the appropriate variables in the file:

Select Language ▼

```

# Keep this file separate
PY4E (https://www.py4e.com)

# https://apps.twitter.com/
# Create new App and get the four strings

def oauth():
    return {"consumer_key": "h7Lu...Ng",
            "consumer_secret": "dNKenAC3New...mmn7Q",
            "token_key": "10185562-eibxCp9n2...P4GEQQ0SGI",
            "token_secret": "H0ycCFemmC4wyf1...qoIpBo"}

# Code: http://www.py4e.com/code3/hidden.py

```

The Twitter web service are accessed using a URL like this:

https://api.twitter.com/1.1/statuses/user_timeline.json (https://api.twitter.com/1.1/statuses/user_timeline.json)

But once all of the security information has been added, the URL will look more like:

```

https://api.twitter.com/1.1/statuses/user_timeline.json?count=2
&oauth_version=1.0&oauth_token=101...SGI&screen_name=drchuck
&oauth_nonce=09239679&oauth_timestamp=1380395644
&oauth_signature=rLK...BoD&oauth_consumer_key=h7Lu...GNg
&oauth_signature_method=HMAC-SHA1

```

You can read the OAuth specification if you want to know more about the meaning of the various parameters that are added to meet the security requirements of OAuth.

For the programs we run with Twitter, we hide all the complexity in the files *oauth.py* and *twurl.py*. We simply set the secrets in *hidden.py* and then send the desired URL to the *twurl.augment()* function and the library code adds all the necessary parameters to the URL for us.

This program retrieves the timeline for a particular Twitter user and returns it to us in JSON format in a string. We simply print the first 250 characters of the string:

```

import urllib.request, urllib.parse, urllib.error
import twurl
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

TWITTER_URL = 'https://api.twitter.com/1.1/statuses/user_timeline.json'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                        {'screen_name': acct, 'count': '2'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()
    print(data[:250])
    headers = dict(connection.getheaders())
    # print headers
    print('Remaining', headers['x-rate-limit-remaining'])

# Code: http://www.py4e.com/code3/twitter1.py

```

When the program runs it produces the following output:

```

Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at": "Sat Sep 28 17:30:25 +0000 2013",
 "id": 384007200990982144, "id_str": "384007200990982144",
 "text": "RT @fixpert: See how the Dutch handle traffic
intersections: http://t.co/tIiVWtEhj4\n#brilliant",
 "source": "web", "truncated": false, "in_rep
Remaining 178

Enter Twitter Account:fixpert
Retrieving https://api.twitter.com/1.1/ ...
[{"created_at": "Sat Sep 28 18:03:56 +0000 2013",
 "id": 384015634108919808, "id_str": "384015634108919808",
 "text": "3 months after my freak bocce ball accident,
my wedding ring fits again! :)\n\nhttps://t.co/2XmHPx7kgX",
 "source": "web", "truncated": false,
Remaining 177

Enter Twitter Account:

```

Along with the returned timeline data, Twitter also returns metadata about the request in the HTTP response headers. One header in particular, `x-rate-limit-remaining`, informs us how many more requests we can make before we will be shut off for a short time period. You can see that our remaining retrievals drop by one each time we make a request to the API.

In the following example, we retrieve a user's Twitter friends, parse the returned JSON, and extract some of the information about the friends. We also dump the JSON after parsing and "pretty-print" it with an indent of four characters to allow us to pore through the data when we want to extract more fields.

```
import urllib.request, urllib.parse, urllib.error
import twurl
import json
import ssl

# https://apps.twitter.com/
# Create App and get the four strings, put them in hidden.py

TWITTER_URL = 'https://api.twitter.com/1.1/friends/list.json'

# Ignore SSL certificate errors
ctx = ssl.create_default_context()
ctx.check_hostname = False
ctx.verify_mode = ssl.CERT_NONE

while True:
    print('')
    acct = input('Enter Twitter Account:')
    if (len(acct) < 1): break
    url = twurl.augment(TWITTER_URL,
                       {'screen_name': acct, 'count': '5'})
    print('Retrieving', url)
    connection = urllib.request.urlopen(url, context=ctx)
    data = connection.read().decode()

    js = json.loads(data)
    print(json.dumps(js, indent=2))

    headers = dict(connection.getheaders())
    print('Remaining', headers['x-rate-limit-remaining'])

    for u in js['users']:
        print(u['screen_name'])
        if 'status' not in u:
            print('    * No status found')
            continue
        s = u['status']['text']
        print('    ', s[:50])

# Code: http://www.py4e.com/code3/twitter2.py
```


Since the JSON becomes a set of nested Python lists and dictionaries, we can use a combination of the index operation and for loops to wander through the returned data structures with very little Python code.

The output of the program looks as follows (some of the data items are shortened to fit on the page):


```
Enter Twitter Account:drchuck
Retrieving https://api.twitter.com/1.1/friends ...
Remaining 14
```

```
{
  "next_cursor": 1444171224491980205,
  "users": [
    {
      "id": 662433,
      "followers_count": 28725,
      "status": {
        "text": "@jazzychad I just bought one .__.",
        "created_at": "Fri Sep 20 08:36:34 +0000 2013",
        "retweeted": false,
      },
      "location": "San Francisco, California",
      "screen_name": "leahculver",
      "name": "Leah Culver",
    },
    {
      "id": 40426722,
      "followers_count": 2635,
      "status": {
        "text": "RT @WSJ: Big employers like Google ...",
        "created_at": "Sat Sep 28 19:36:37 +0000 2013",
      },
      "location": "Victoria Canada",
      "screen_name": "_valeriei",
      "name": "Valerie Irvine",
    }
  ],
  "next_cursor_str": "1444171224491980205"
}
```

```
leahculver
  @jazzychad I just bought one .__.
_valeriei
  RT @WSJ: Big employers like Google, AT&T are h
ericbollens
  RT @lukew: sneak peek: my LONG take on the good &a
halherzog
  Learning Objects is 10. We had a cake with the LO,
scweeker
  @DeviceLabDC love it! Now where so I get that "etc
```

Enter Twitter Account:

Select Language ▼

The last bit of the output is where we see the for loop reading the five most recent “friends” of the `@drchuck`  **PY4E** (<https://www.py4e.com>) Twitter account and printing the most recent status for each friend. There is a great deal more data available in the returned JSON. If you look in the output of the program, you can also see that the “find the friends” of a particular account has a different rate limitation than the number of timeline queries we are allowed to run per time period.

These secure API keys allow Twitter to have solid confidence that they know who is using their API and data and at what level. The rate-limiting approach allows us to do simple, personal data retrievals but does not allow us to build a product that pulls data from their API millions of times per day.

If you find a mistake in this book, feel free to send me a fix using Github .