ICS 32 Spring 2015 | [News](http://www.ics.uci.edu/~thornton/ics32/index.html) | [Course Reference](http://www.ics.uci.edu/~thornton/ics32/CourseReference.html) | [Schedule](http://www.ics.uci.edu/~thornton/ics32/Schedule.html) | [Project Guide](http://www.ics.uci.edu/~thornton/ics32/ProjectGuide) | [Code Examples](http://www.ics.uci.edu/~thornton/ics32/CodeExamples) | [About Alex](http://www.ics.uci.edu/~thornton/)

**ICS 32 Spring 2015  
Code Example: Using Web APIs**

***Background***

In our previous [code example](http://www.ics.uci.edu/~thornton/ics32/CodeExamples/index.html), we saw that the Python standard library makes it easy to download a web page. Given a URL, which specifies the location of the page you want to download, the**urllib.request.urlopen()** function hides nearly all of the details of the conversation between your Python program and the web server that will send the page back to it. You do have to know what HTTP is, and you do have to understand a few details of how it works (e.g., status codes, URLs, and perhaps headers), but you don't have the implement the HTTP protocol from scratch, you don't have to know the precise format of an HTTP request or response, and the function more or less "just works," as long as the page is available, your Internet connection is active, and you open the right URL.

However, writing a program to download a web page — the same page you might see if you visited a particular URL in a web browser — often isn't the right choice. A web page is suitable for display to a human user within a browser, but isn't necessarily a suitable result for a program to download. For example, you can visit YouTube and search for whatever you'd like — say, **lakers clippers**, if you like basketball — and you'll get back a result like this one:

* <http://www.youtube.com/results?search_query=lakers+clippers>

But if you wanted to write a Python program that was capable of obtaining just the titles of the videos on that page, you'd have a surprisingly difficult time doing it. The problem is that the web page isn't what it first seems. Its structure looks simple and straightforward when displayed in a browser, but it's actually surprisingly complex. If you've never looked at the source code of a web page before, visit the YouTube link above, then right-click in an empty area of the page and select something like **View Page Source** — it'll be slightly different depending on what browser you use, but will likely be available in any browser running on a laptop or desktop machine, though perhaps not on a smartphone or other small-sized device. What you'll see is a combination of HTML, JavaScript, CSS, and other code used to make the page look the way it does. When your Python program downloads that page, this source code is what your program will get back; finding the important parts — this is sometimes called *web scraping* — is a challenge, one made worse by the fact that YouTube will make changes to its pages periodically that will render your scraping algorithm incompatible, and you'll be back to square one again.

**Using web APIs instead of web pages**

Fortunately, some sites provide an alternative interface, one that's intended to be used by programs instead of people. These are sometimes called *APIs* (application programming interfaces), because their goal is to provide an interface that application programs can use to access them. (Note that the term "API" is actually a pretty open-ended one, used not only to describe interfaces to web sites, but also any other kind of software library; here, though, when I say "API," I'm referring to a web API.)

YouTube is owned by the tech giant Google. In addition to providing YouTube as a web site that you can visit normally in a browser, Google also provides a set of APIs that can be used to access information about YouTube videos, channels, playlists, and users; to upload new videos to your YouTube account; to view an analysis of things like advertisement sales on your own videos; and so on. Some of what's provided requires payment; other parts are free. While a fair amount of what they provide is only available if you *authenticate* (i.e., you're gone through a procedure to log into the API using your Google account, which is beyond the scope of what we're doing here), some of the more innocuous functionality — such as finding videos that match a search query — is available to anyone, subject only to reasonable usage limits (i.e., how many times per day you can run a query).

Since our goal here is to display titles and descriptions of videos matching a YouTube search query, we'll need an API called the [YouTube Data API](https://developers.google.com/youtube/v3/). The YouTube Data API is a web API, meaning that a program interacts with it by sending an HTTP request — just like downloading a web page — and gets its answer back as an HTTP response. The URL specifies not only the operation we want to perform (e.g., search), but also the parameters for that operation (e.g., the search query). Meanwhile, the response is formatted in a way that's structured so that it will be easy for a program to parse and understand, in a format that's published (it's part of the API), so you can rest assured that it won't change when YouTube periodically changes the look of their web pages for human users.

Using web APIs effectively requires us to learn a handful of new techniques, though we'll find that most of them (certainly YouTube's) are implemented around *standards*, in ways that are common among most web APIs. These techniques are so common that Python's standard library provides implementations of all of them, so we don't have to deal with any of the low-level details; we just have to know what parts of the library we need, and we'll be in business. To find what we need in the library, we first have to learn a little bit about the standards techniques that are being used.

***Standards for web APIs***

While there are certainly differences between web APIs, and there are common techniques that we won't need here, the YouTube Data API uses a few standard techniques that we'll need to be familiar with.

**URLs with query parameters**

Since the very early days of the World Wide Web, it has been necessary to specify URLs that carry parameters, particularly on web sites whose content is generated dynamically. For many years, there has been a standard for URLs that include these kinds of parameters, called *query parameters*. A hypothetical example of a URL with query parameters follows:

* http://www.blah.com/some/page?param1=value1&param2=value2&param3=value3

Before the **?** character, this looks just like any other URL. The **?** is special; it indicates that what follows it will be a sequence of query parameters. Each parameter is specified as a name and a value, with an **=** separating them; the parameters themselves are separated by **&** characters.

For example, I opened Amazon.com in my browser and searched for **u2 the joshua tree** and here's the URL that my browser directed me to:

* http://www.amazon.com/s/ref=nb\_sb\_noss\_1/188-0999242-2980934?url=search-alias%3Daps&field-keywords=u2+the+joshua+tree

We can see that the parameters here are **url** (whose value is some kind of URL, though it's not clear exactly what it's being used for) and **field-keywords** (which appears to be my original query, with the spaces mysteriously replaced with **+** characters). We would have to know more about how Amazon's web site is implemented to know for sure what the query parameters mean, but we can sometimes suss out their meaning just by looking at them.

**URL encoding**

So, in general, the syntax for URLs with query parameters is quite simple. After the **?**, parameters are separated by **&** characters, with the name and value of each parameters separated by **=**characters. But who do we specify a parameter whose value contains an **=** character? How about a parameter whose name includes a **?** character? Or a parameter whose value includes an **&**character? Or a parameter whose value includes spaces (which turn out to be illegal in URLs)?

The answer lies in a technique called *URL encoding*, in which any character that's considered "special" (i.e., it has a meaning in the syntax, like **=** or **&**) is replaced by something else. There are two main rules:

* Spaces are replaced with **+** characters.
* Other special characters — anything that isn't legal in a URL, or has a special meaning (such as **?**, **&**, or **=**) is replaced by a **%** followed by a "character code" (a numeric code, expressed in *hexadecimal* (i.e., base 16) corresponding to the character). For example, a **?** is replaced with **%3F**.

(A much more detailed explanation of this kind of encoding is [here](http://en.wikipedia.org/wiki/Percent-encoding), if you're curious about it, though there's no need for you to understand it any better than I've explained it here. The basic idea is sort of the same as escape sequences in Python string literals, which allow us to include characters in our string literals — like quotes or newlines — that are otherwise syntactically problematic.)

We do need to be able to recognize whether something has been URL encoded — which we can see by looking for **%** characters followed by two hexadecimal digits — but the details will be left to part of Python's standard library to implement for us.

**JavaScript Object Notation (JSON)**

The YouTube Data API, like many web APIs, returns its results in a common format called**JavaScript Object Notation** — usually referred to by its acronym, **JSON**. Because many web APIs are consumed by web pages, and the "guts" of many web pages are written in a programming language called JavaScript, JSON (which has its roots in JavaScript) turns out to be a natural format for web APIs to return. And JSON has become so common in web APIs and across the Internet, the Python standard library provides a module that can parse it, so we won't have to.

JSON is actually quite simple. An example follows:

{ "name": "Boo", "age": 9, "qualities": ["intelligent", "cute", "playful", "perfect"] }

This is a JSON description of an *object*, which consists of a collection of *attributes* (much like Python's objects do). In this example, the attribute **name** has the string value **"Boo"**, the attribute**age** has the numeric value **9**, and the attribute **qualities** has a value that is an array (you can think of these like Python lists) containing the strings **"intelligent"**, **"cute"**, **"playful"**, and **"perfect"**.

The details of JSON are described at [json.org](http://json.org/), but they're not much more complicated than that.

***Using the YouTube Data API***

The YouTube Data API allows us to send a wide variety of different kinds of requests, but we'll focus on just one for this example. Our goal is to issue a search query — like we might do on YouTube's web page — and display the titles and descriptions of videos that match the request.

The appropriate request in the YouTube Data API is called **search**, which is described in detail here:

* <https://developers.google.com/youtube/v3/docs/search/list>

Boiling down their documentation, we need to know only a few things, though you might want to look through and see what else is available, in case you want to experiment with aspects of it that we're not covering here.

* A **search** request is issued using a URL that starts out like this:
  + **https://www.googleapis.com/youtube/v3/search**
* Following the word **search** in the URL, we add a **?** character and then include a list of query parameters. There are lots of parameters we might like to pass, but these are the ones we need in order to accomplish our goal:
  + **key**, which is an *API key* that uniquely identifies us as a user of Google's web APIs. We're required to set up an API key and associate with our Google account if we want to use Google's APIs, though I've already done that for us (and I'll send out an API key in an email message). You can feel free to create your own, too, if you'd like, by logging into your Google account and then visiting the [Google Developers Console](https://developers.google.com/console/).
  + **part**, which specifies what part of YouTube's information you're interested in seeing. In our case, we want something called a **snippet**.
  + **type**, which specifies what we want to search for. In our case, we want to search for**video**, though we could also search for other things (**channel** or **playlist**) instead.
  + **maxResults**, which specifies how many results we want (at most).
  + **q**, which specifies our search query.

So a complete URL might look like this:

* https://www.googleapis.com/youtube/v3/search?key=***YOUR\_API\_KEY***&part=snippet&type=video&maxResults=10&q=lakers+clippers

Of course, we need to be sure that the parameters are URL encoded in case, for example, the search query includes special characters in it. Note the **+** sign between **lakers** and **clippers** in the example above; that's because the search query contained a space, but spaces are URL encoded to **+** signs.

We then issue the request — an HTTP request, just like we've seen before, albeit one that includes encryption (which is why the URL starts with **https** instead of **http**) — and the result is returned as a JSON object.

The only remaining trick is to understand what parts of the JSON response we're interested in. Construct a URL using the pattern above (replacing ***YOUR\_API\_KEY*** with a valid Google API key, such as the one I will email to you) and visit it in a browser, then take a look through the JSON response to see if you can understand what parts of it you might need, given our goal to display the titles and descriptions of the matching videos.

***How the Python standard library can help***

There are three basic operations we need here:

* To URL encode a set of query parameters, so that it's safe to pass them to a web API.
* To issue an HTTP request to the URL we built and get the HTTP response.
* To parse the response, which will be a JSON object in the format documented in the YouTube Data API documentation.

**URL encoding your query parameters**

The module **urllib.parse**, which sounds like it knows how to parse (i.e., read) URLs, happens also to contain a function that knows how to URL encode query parameters. The easiest way to use it is to pass it a list of two-element tuples, and it will generate URL-encoded query parameters from it. For example:

>>> **import** urllib.parse

>>> urllib.parse.urlencode([('name', 'Boo'), ('age', 9), ('description', 'pekingese/perfect'), ('search', 'lakers clippers')])

'name=Boo&age=9&description=pekingese%2Fperfect&search=lakers+clippers'

**Issuing an HTTP request and getting the HTTP response**

This is actually no different than what we did in the previous [code example](http://www.ics.uci.edu/~thornton/ics32/CodeExamples/index.html). We use the**urllib.request.urlopen** and pass it our full URL, including the URL-encoded parameters, then decode the response bytes into a string, just like we did before.

**Parsing the JSON response**

Once we've retrieved a string from the HTTP response, we'll have the response text in JSON format. But a string in JSON format isn't a very convenient thing to have; to process it, we'll need to start searching for curly braces, double quotes, commas, colons, brackets, etc., in order to build an understanding of what's there and act on it. If JSON is really intended as a way to describe an object, and if Python also implements that same notion — Python objects have attributes that have names, each with a value that can be of any type you'd like — then what we'd really like to do is take the JSON string and turn it into a Python object.

That would be a tall task, except for one important bit of good news: the Python standard library includes a module called **json** that knows how to do this already! It can turn JSON strings into Python objects (and also Python objects back into JSON strings again, though we won't need that here). For example:

>>> **import** json

>>> x = '{ "name": "Boo", "age": 9, "qualities": ["intelligent", "cute", "playful", "perfect"] }'

>>> obj = json.loads(x)

>>> obj['name']

'Boo'

>>> obj['age']

9

>>> **for** quality **in** obj['qualities']:

**print**(quality)

intelligent

cute

playful

perfect

Now *that's* handy, isn't it?

***The code***

Bringing all of this together, below is the program that we wrote in lecture that uses the YouTube Data API to display information about YouTube videos that are relevant, given a search phrase such as **lakers clippers**. The program requires an *API key* from Google. I created one for this course, which I will send out separately via email; you can also create your own via the [Google Developers Console](https://developers.google.com/console/) and use that, if you prefer.

* [**youtube.py**](http://www.ics.uci.edu/~thornton/ics32/CodeExamples/WebAPIs/youtube.py)