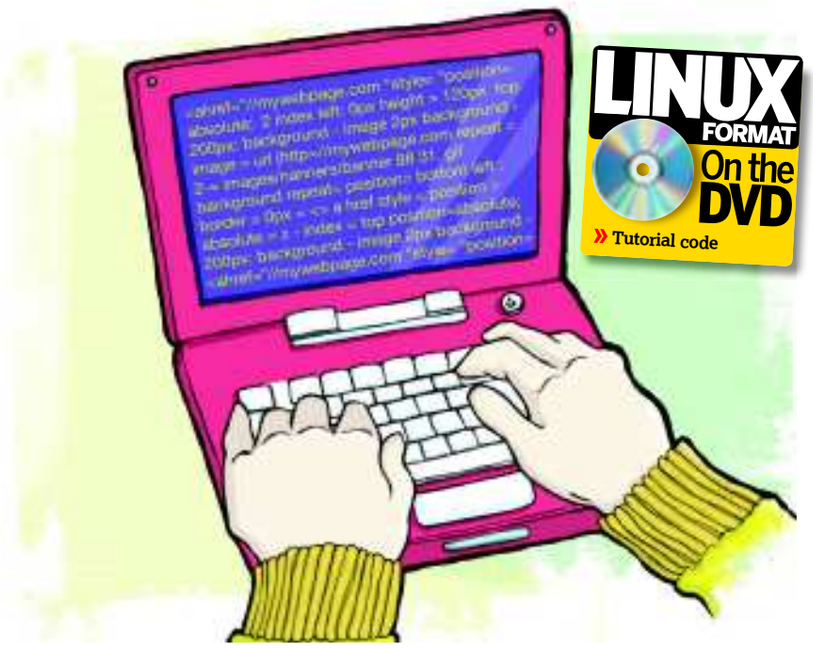
Python: Digg

Part 4: We unlock the web's APIs with the power of XML. Your intrepid guide to all things Python, **Nick Veitch**, won't rest until he has dug Digg.



Our expert

Nick Veitch launched and edited *Linux Format* for its first eight years. Then he went bad.

Previously in these tutorials we've used existing API code to interface with our web objects of desire and turn them to our needs. This saves us a lot of work, but it's also somewhat restrictive and leaves us dependent on others. And anyway, why should we have to bother importing an entire API module if all we want are a few methods?

So this time, even though there's a perfectly reasonable – if slightly out of date – API for Digg, we're going to create our own instead. Well, sort of. It isn't going to cover all of the functionality of the Digg site, but it will demonstrate enough that you could go on to create a full API if you wanted to.

› The Digg API has its own site and plenty of documentation, although it isn't always that easy to follow.

Many sites use a particular form of communication for their API and some use several. Of these, the most popular are JSON and XML. JSON is simpler and easy to integrate into JavaScript, which is why it often pops up. XML is bigger, chunkier and prettier. It's also slightly easier to follow, but you end up with swathes of response even for a simple query. Still, it's what we'll use here.

To be fair, there isn't a great deal of difference between them, but XML is the *lingua franca* of the web, so if you can handle API calls through XML, it will stand you in good stead.

Python handles XML well and has an established module for manipulating it, but we're getting ahead of ourselves. The first thing we need to do is determine how we can interact with the Digg API in the first place. And, as with many web-savvy social websites, Digg has lots of documentation for programmers on how to use the API. Yay!

Digg it URL style

Head over to the Digg API site (**http://apidoc.digg.com**) and take a look at what's on offer. The API is accessed through the main website via queries. You're probably familiar with these already – they're a collection of values that are passed to the webserver for processing, beginning with a **?** and separated by **&**. We don't even need to write a single line of code ourselves to try them out, because we can just make up a URL and type it into a browser. Since we're going to use the default XML response, a browser such as *Firefox* will display the resulting XML code directly in the browser window without having to save it to a file first, which is a great boon. After all, the only thing more annoying than typing in a complicated request and generating an error is having to load up the file in your text editor to find out that it didn't work.

The Digg API works by having specific endpoints, or URL paths, to activate particular functions. So, for example, if you want to find out the latest hot list you would navigate to **http://services.digg.com/stories/hot**.

Try it in your browser and you'll get an error message, though. The only proviso from Digg to provide these services is that the application making the request needs to have an appkey (or API key). This is common among web services – they want to be able to identify particular clients. This isn't for nefarious reasons, but if something is misbehaving and hitting the server every 10 milliseconds, they want to be able to block it without bringing down the whole API.

In this respect, Digg is a little unusual, because it doesn't require that you register an API key in advance, just that you provide a URL to your application or the source of your client

» **Last month** We made Twitter spit your tweets using Python. Lovely.

LXF123.tut_python 94 30/7/09 4:06:43 pm

through XML

software. So, if we try again with the appkey **http://services.digg.com/stories/hot/?appkey=http://linuxformat.co.uk**, you should see a screen full of structured text. The **?** In the URL indicates a query and we are going to post some values. The usual format for this is a list of key=value pairs, separated by an ampersand, **&**. If we now change our URL to **http://services.digg.com/stories/hot/?appkey=http://linuxformat.co.uk&count=1** then this time we will only see the very first story. Obviously you could supply a different count, or any of the other variables this particular query accepts. How do you know what they are? Well, you have to rely on Digg to publish them. In this case, you can find the arguments for the endpoint towards the bottom of **http://apidoc.digg.com/ListStories**.

Digg with Python

So, now we know the basics of the Digg API. We can type in a URL and get it to return some XML in exchange. That's all very well, but how do we deal with this programmatically? Well, the first thing we need in this case is a way to open a URL. The standard Python module, *urllib*, can do this for us, so let's fire up a Python shell (open up a terminal window and just type **python**) and see what we can do.

```
>>> url='http://services.digg.com/stories/hot/?'
>>> appkey='http:///linuxformat.co.uk'
>>> import urllib
>>> diggargs ={ 'count': 1, 'appkey': appkey}
>>> foo=urllib.urlencode(diggargs)
>>> request = url+foo
>>> request
'http://services.digg.com/stories/hot/?count=1&appkey=http%3A%2F%2Flinuxformat.co.uk'
>>>
```

OK, this needs a little bit of explanation. First up, we store our base URL and our appkey as convenient strings, because they'll get used a lot. Once we've imported the *urllib* module, we want to set up some variables to pass to Digg. Here we've used a Python dictionary. This is a simple construct contained in curly brackets that takes key=value pairs and behaves a little like a list. The name of the value comes first, followed by a colon and the actual value. The values can be any valid Python type, but here they're likely to take the form of either strings or integers.

Why use a dictionary?

Why did we bother creating a dictionary out of our arguments? As well as for the sake of being nice and neat, there's a helper function in *urllib* that will turn these into a query string for us. This isn't as simple as just concatenating – joining together – all the strings, because HTML has rules »

Quick tip

Beginners often get stuck in the Python shell on the terminal because the standard Ctrl+C shortcut doesn't work. To quit the Python shell, type Ctrl+D instead.

› **If you use *Firefox* or a similar browser, it will show you the XML that's returned from a query in a default, structured way.**

» **If you missed last issue** Call 0870 837 4773 or +44 1858 438795.

LXF123.tut_python 95

30/7/09 4:06:44 pm

» about the characters that can be sent as requests. So, the next line employs the helper function **urllib.urlencode** to translate our dictionary into a query string. Another advantage of using a dictionary is we can easily add or change values and regenerate the query string. By contrast, it would be tricky to make changes if we'd just converted our arguments to a query string directly.

A massive response

The request is built just by joining the base URL and the query string. If you're curious, you can just type in this variable name and Python will output its value – in this case a barely comprehensible URL, which is the result of the correct encoding. So, what do we get if we connect to the server with that request? Well, it should look like this:

```
>>> response = urllib.urlopen(request)
>>> response.read()
'<?xml version="1.0" encoding="utf-8" ?>\n<stories
timestamp="1247411540" total="12257" offset="0"
count="1">\n <story link="http://www.thedailybeast.com/
blogs-and-stories/2009-07-11/young-gop-chooses-hate/"
submit_date="1247350492" diggs="109" id="13923829"
comments="46" href="http://digg.com/politics/Young_GOP_
Chooses_Hate" status="upcoming" media="news">\n
<description>Audra Shay, the Young Republican leader
accused of endorsing racism on Facebook, was elected head
of the group for GOP members under 40 this afternoon.</
description>\n  <title>Young GOP Chooses Hate </title>\n
<user name="Pash1994" registered="1220537298"
profileviews="2218" icon="http://digg.com/users/Pash1994/l.
png" />\n  <topic name="Political News" short_
name="politics" />\n  <container name="World &amp;
Business" short_name="world_business" />\n  <thumbnail
originalwidth="174" originalheight="174"
contentType="image/jpeg" src="http://digg.com/politics/
Young_GOP_Chooses_Hate/t.jpg" width="80" height="80"
/>\n  <shorturl short_url="http://digg.com/d1wQDt" view_
count="309" />\n </story>\n</stories>'
```

The **urlopen** function returns a file-like object that can be manipulated like any other file object. This can be useful if you're expecting a huge amount of data, but I doubt you're going to run out of memory in our Digg experiments. So, **response.read()** will just dump the file for us, and you might want to stick these two together like so:

```
>>> response = urllib.urlopen(request).read()
```

This is a trick you can use with most Python objects, although it can make the code harder to understand.

Playing with XML

In the example here, we have pretty short and straightforward XML that doesn't descend too far. For larger documents, and if you want to start generating XML yourself, you should install a good editor.

There are a few XML-specific editors such as *XMLCopyEditor*, which should keep you right when it comes to your structure. They'll also make it easier to scan and search through the text.

› An XML editor can save you time, grief and embarrassment.

Our cunning trick has worked and we now have our response – a huge mass of XML to deal with. To parse it in as an XML object, we need to invoke some methods from Python's XML module as follows:

```
>>> from xml.dom import xml.minidom
>>> x = minidom.parseString(response)
```

Collecting data from objects and constructing a well-formed XML file from that information is known in XML parlance as marshalling. What we want to do, however, is the reverse – create a Python object from the data. What follows is probably one of the most copied bits of code ever, at least in terms of Python and XML. It originates, I believe, in some code written by Mark Pilgrim (the author of *Dive Into Python*), but feel free to disagree via the usual address...

```
class Bag: pass

def unmarshal(element):
    rc = Bag()
    if isinstance(element, minidom.Element):
        for key in element.attributes.keys():
            setattr(rc, key, element.attributes[key].value)

    childElements = [e for e in element.childNodes \
                     if isinstance(e, minidom.Element)]
    if childElements:
        for child in childElements:
            key = child.tagName
            if hasattr(rc, key):
                if type(getattr(rc, key)) <> type([]):
                    setattr(rc, key, [getattr(rc, key)])
                setattr(rc, key, getattr(rc, key) + [unmarshal(child)])
            elif isinstance(child, minidom.Element) and \
                    (child.tagName == 'Details'):
                # make the first Details element a key
                setattr(rc,key,[unmarshal(child)])
            else:
                setattr(rc, key, unmarshal(child))
    else:
        text = "".join([e.data for e in element.childNodes \
                        if isinstance(e, minidom.Text)])
        setattr(rc, 'text', text)
    return rc
```

This will also take a little bit of explaining. The first odd thing here is the **Bag** class, which seems to have been defined with nothing in it. This is permissible in Python – indeed, how could we know what was going to be in the class until we've unpacked the data? It also demonstrates the flexibility of Python perfectly; it enables you to have a class of object that you make up as you go along.

Code anarchy?

The **unmarshal** function is simply a recursive procedure that steps down every node of the XML DOM and creates a Python object out of it. If you wanted to create an API module for Python to interpret the Digg output, you could do so in a more structured and meaningful way, because you would know the structure of the XML. This method is a sort of global catchall, with the downside that the object you created is still ungainly. However, some post-processing can sort that out.

What we have is a list of **story** containers, which are themselves held inside a container called **stories**. Each has its own subnodes for comments, IDs, URLs and so on. If we

» **Never miss another issue** Subscribe to the #1 source for Linux on p102.

LXF123.tut_python 96 30/7/09 4:06:44 pm

wanted to take a look at the **story** objects, we could just iterate over the **stories** container like this:

```
>>> for item in bar.stories.story:
>>>    print item.id, item.link,  item.title.text
```

There's no reason why we shouldn't programmatically add data to this structure as well. What if, for instance, we wondered where all these stories were being published? Well, there's a useful free library called *GeoIP* that matches IP addresses to countries.

Adding more data

The *GeoIP* module is available for major distros, or you can download it from MaxMind (**www.maxmind.com/app/python**). It's quite simple – you create a **GeoIP** object, then use its methods to resolve a country code or name from the domain name of a web server. Here's a quick demo:

```
>>> import GeoIP
>>> geo=GeoIP.new(GeoIP.GEOIP_STANDARD)
>>> geo.country_name_by_name('google.com')
'United States'
```

Pretty simple stuff. Unfortunately, it only wants the domain, not the whole URL. However, we can import yet another module from the standard Python modules, called *urlparse* (**www.python.org/doc/2.5.2/lib/module-urlparse.html**), which will split up a URL into bits for us.

```
>>> import urlparse
>>> for item in bar.stories.story:
>>>    item.country=geo.country_name_by_name(urlparse.
urlparse(item.link).netloc)
>>>    print item.country
```

Here we've rewritten our loop and inside that we're creating a new property of our **item** object called **country**. Feeding the chopped-up domain name from the **urlparse** function into our **GeoIP** function spits out the name of the country as a string.

If we wanted to go a bit further, we could take the country values recorded and make a dictionary out of their frequency in, say, the top 100 hot sites. This is pretty easy to do. We just declare an empty dictionary then add one to the value of that country's key as we loop through the stories. If that key doesn't exist, we use a default value of zero. The dictionary can be translated into a sorted list at the end and you could easily draw a bar chart of country frequency.

```
#!/usr/bin/python

import urllib
from xml.dom import minidom
import urlparse,  GeoIP,  operator

url='http://services.digg.com/stories/hot/?'
appkey='http://linuxformat.co.uk'
geo=GeoIP.new(GeoIP.GEOIP_MEMORY_CACHE)


class Bag: pass

def unmarshal(element):
    rc = Bag()
    if isinstance(element, minidom.Element):
        for key in element.attributes.keys():
            setattr(rc, key, element.attributes[key].value)
    childElements = [e for e in element.childNodes \
                if isinstance(e, minidom.Element)]
    if childElements:
        for child in childElements:
            key = child.tagName
            if hasattr(rc, key):
                if type(getattr(rc, key)) <> type([]):
                    setattr(rc, key, [getattr(rc, key)])
                setattr(rc, key, getattr(rc, key) + [unmarshal(child)])
            elif isinstance(child, minidom.Element) and \
                    (child.tagName == 'Details'):
                # make the first Details element a key
                setattr(rc,key,[unmarshal(child)])
            else:
                setattr(rc, key, unmarshal(child))
    else:
        text = "".join([e.data for e in element.childNodes \
                    if isinstance(e, minidom.Text)])
        setattr(rc, 'text', text)
    return rc


diggargs ={ 'count': 100, 'appkey': appkey}
foo=urllib.urlencode(diggargs)
request = url+foo
response = urllib.urlopen(request).read()

x = minidom.parseString(response)
bar = unmarshal(x)

hist = {}
for item in bar.stories.story:

    item.country=geo.country_name_by_name(urlparse.
urlparse(item.link).netloc)
    hist[item.country]=hist.get(item.country, 0) +1

sorted = sorted(hist.items(), key=operator.itemgetter(1),
reverse=True)
print sorted
```

We've covered a lot of ground here, even though we've only tackled one Digg endpoint. For a more useful API, you'd want to build a class and a few objects to cover users, stories and so on, using the tricks here to populate them. Digg is mostly one-way traffic, but next time we'll be looking at how to write data too as we build a GUI Flickr client. LXF

› **If you need more help learning Python, try the excellent *Dive Into Python* in print or online.**

Quick tip

If you're experimenting in Python 3 you will discover that *urllib* no longer works. That's because it has been split into parts for Python3, *urllib.request*, *urllib.parse* and *urllib.error*. You can get more info at the python docs site **http://docs.python.org/library/urllib.htm**.

» **Next month** We're getting GUI as we build a standalone Flickr app.

LXF123.tut_python 97

30/7/09 4:06:46 pm