“R” is for Replacement

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). He is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

As Python programmers know, there has always been a “batteries included” philosophy when it comes to Python’s standard library. For instance, if you simply download Python and install it, you instantly get access to hundreds of modules ranging from XML parsing to reading and writing WAV files.

The standard library is both a blessing and a curse. Because of it, many programmers find they can simply install Python and have it work well enough for their purposes. At the same time, reliance on the library and concerns about backwards compatibility tend to give it a certain amount of inertia. It is sometimes difficult to push for changes and improvements to existing modules. This is especially true if one tries to challenge the dominance of standard library modules for extremely common tasks such as regular expression parsing or network programming.

In this article, I’m going to take a brief tour through two third-party libraries, requests and regex, that have generated a bit of buzz in the Python world by aiming to replace long-standing and widely used standard library modules. Both have generated buzz in the Python world and, coincidentally, both start with the letter “R.”

Interacting with the Web

Python has long included a module, urllib, that gives you simple access to the Web. For example, if you want to download and print out the street address of every bike rack in the city of Chicago, you can write code like this:

```
import urllib
u = urllib.urlopen("http://data.cityofchicago.org/api/views/cbyb-69xx/rows.
csv")
for line in u:
        fields = line.split(",")
        print fields[1]
```

This works fine if all you want to do is pull down a simple document and read it. However, as you know, the Web is a complicated place. If you need to do almost anything else, such as supply custom HTTP headers, provide form data, upload files, perform authentication, or deal with cookies, you’re out of luck.

Some limitations of urllib are addressed by another standard library, creatively named urllib2. However, if you’ve ever used urllib2 you know that it feels “over engineered” and that seemingly simple tasks like authentication can be tricky. To give you some idea, here is a fragment of code that shows how you would initiate a basic authentication login to the Python Package Index (http://pypi.python.org).

```
import urllib2
auth = urllib2.HTTPBasicAuthHandler()
auth.add_password("pypi","http://pypi.python.org","username","password")
opener = urllib2.build_opener(auth)

r = urllib2.Request("http://pypi.python.org/pypi?:action=login")
u = opener.open(r)
resp = u.read()

# From here. You can access more pages
```

As you can see, the process has become quite a bit more complicated. It gets far more convoluted, if not practically impossible, if you want to do anything more advanced, such as invoke other HTTP methods (e.g., HEAD, PUT, DELETE, etc.), upload files, or read streaming data.

Although you could continue to hack away on urllib2 to try to make it do what you want, you might be better off looking at Kenneth Reitz's requests library instead (http://pypi.python.org/pypi/requests). Rather than trying to emulate existing functionality, requests provides an entirely different programming interface.

First, let's just download a simple Web page:

```
>>> import requests
>>> r = requests.get("http://www.python.org")
>>> r.status_code
200
>>> r.headers
{'last-modified': 'Fri, 27 Jan 2012 15:49:35 GMT',
  'content-length': '18882',
  'etag': '"105800d-49c2-4b7847185c1c0"',
  'date': 'Sat, 28 Jan 2012 19:13:11 GMT',
  'accept-ranges': 'bytes',
  'content-type': 'text/html',
  'server': 'Apache/2.2.16 (Debian)'
}
>>> page = r.text
>>> page
u'<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"...'
>>>
```

That was certainly easy enough. Although it looks easy, there are some subtle things going on under the covers. First, access to the r.text attribute automatically performs the appropriate Unicode decoding, returning a Unicode string. Thus, you don't have to worry about it. Similarly, you can easily obtain status code and header information as shown.

Now, let's make a HEAD request to see if the document has changed recently:

```
>>> r = requests.head("http://www.python.org")
>>> r.headers
{'last-modified': 'Fri, 27 Jan 2012 15:49:35 GMT',
  'content-length': '18882',
  'etag': '"105800d-49c2-4b7847185c1c0"',
  'date': 'Sat, 28 Jan 2012 19:23:08 GMT',  'accept-ranges': 'bytes',
  'content-type': 'text/html',
  'server': 'Apache/2.2.16 (Debian)'
}
```

```
>>> r.text
u''
>>>
```

That was also easy. Believe it or not, this is practically impossible to do using urllib or urllib2, since they don't provide an interface for changing the HTTP method.

Let's look at an example of authentication. This example shows how to log in to the Python Package Index using basic authentication as shown earlier:

```
import requests
r = requests.get("http://pypi.python.org/pypi?:action=login",
                      auth=("user","password"))
resp = r.text
```

If you wanted to know whether any cookies were set on a page, simply access the cookies attribute:

```
>>> r = requests.get("http://pypi.python.org/pypi?:action=login",auth=("user
","password"))
>>> r.cookies
{'pypi': '0bf0722dee2203ee3accf4fef9650b2f'}
>>>
```

To pass cookies back on subsequent requests, simply supply the r.cookies dictionary as an argument:

```
>>> r2 = requests.get(newurl, cookies=r.cookies)
>>>
```

Let's write a request that reads real-time data from Twitter's streaming API for anything that mentions the word "python". You'll need to supply your own username and password for this:

```
import requests
import sys
import json
url = "https://stream.twitter.com/1/statuses/filter.json"
parms = {
    'track' : 'python',
    }

auth = ('username','password')
r = requests.post(url, data=parms, auth=auth)
for line in r.iter_lines():
    if line:
        print json.loads(line)
```

Here, requests will open a connection and simply feed you a stream of lines as they are produced. Again, it's relatively straightforward. However, don't even try it with urllib2. There is far more that you can do with requests, but this should have given you a small taste for it.

Regular Expression Pattern Matching

The standard library module for handling regular expression parsing is re. If I recall correctly, it is the second implementation of regular expressions, first appearing about 14 years ago in Python 2.0. However, just when you thought re might be the last word in Python regular expression handling, a new library called regex

has appeared. regex is the work of Matthew Barnett and has recently been officially blessed for inclusion in the standard library starting with Python 3.3 (not yet released). However, you can use it now if you simply download it from http://pypi.python.org/pypi/regex. (Editor's Note: You may not be able to install regex on top of versions of Python older than 2.6.4.)

regex is a drop-in replacement for the standard re library. Thus, any regex matching code that you might have written before should still work. An easy way to try regex without making too many changes is to simply change the import statement as follows:

```
import regex as re

# Use re library as before
...
```

The new regex library fixes a huge number of issues, annoyances, and bugs related to the old re library. These include various convenience features, such as showing you the pattern when pattern objects are inspected or printed, as here:

```
>>> pat = regex.compile("[a-zA-Z_][a-zA-Z0-9_]+")
>>> pat
regex.Regex('[a-zA-Z_][a-zA-Z0-9_]+', flags=regex.V0)
>>>
```

You also get a much simplified way to refer to capture groups.

```
>>> pat = regex.compile(r"(\d+)/(\d+)/(\d+)")
>>> m = pat.match("1/29/2012")
>>> m[0]
'1/29/2012'
>>> m[1]
'1'
>>> m[2]
'29'
>>> m[3]
'2012'
>>> m[1:]
('1', '29', '2012')
>>>
```

Behind the scenes, limitations related to the number of capture groups, concurrent operation with threads, and other matters are fixed, in addition to a number of tricky issues with Unicode (e.g., proper case folding).

However, besides subtle cosmetic and implementation improvements, regex offers an interesting range of new functionality. There are too many additions to cover in their entirety, but let's look at a few of the more interesting enhancements.

Suppose you wanted part of a regex to match a set of possible words or symbols. For example, suppose you wanted to match some of Python's math operators (+, -, *, /, and **). You might be inclined to write a regex like this:

```
import regex
pat = regex.compile(r'\*\*|\*|\+|-|/')
```

However, if you look at such a pattern, there are all sorts of tricky escapes (for * and +). Plus, you have to worry about matching in the right order (checking ** prior to *). Here is an alternate approach using named sets:

```
import regex
ops = { '+', '*', '-', '/', '**' }   # A set of everything you want to match
pat = regex.compile('\L<ops>', ops=ops)
```

In this version, you don't have to worry about escaping any of the possible matches or their order. You simply pass a set using the \L escape and specify an appropriate keyword argument (which contains a list or set of the literal symbols you want to match). regex will figure everything out, including the escaping and ordering, to make it work.

If you have written regular expressions, you probably know about the specification of character sets such as [a-zA-Z] or [^a-zA-Z]. regex takes this much further by allowing common set operations (union, intersection, difference), as well as an interface to the Unicode properties. Thus, you can start writing character set patterns like this:

```
# Match all letters except vowels
pat1 = regex.compile("[[a-z]--[aeiou]]+$",regex.V1)
pat1.match("xyzzy")     # Matches
pat1.match("plugh")     # Doesn't match

# Match any non-ascii character
pat2 = regex.compile(r"[^\p{ASCII}]+",regex.V1)
pat2.search(u"That's a spicy jalape\u00f1o")  # Matches
pat2.search(u"I want another torta")          # No matches
```

Finally, another interesting feature is support for fuzzy matching. This is a matching technique where text with errors in the form of insertions, deletions, or substitutions can be matched. Here is an example:

```
>>> pat = regex.compile("(?:spam){s<=1}")
>>>
```

This regex pattern specifies that the text "spam" is to be matched, but that, at most, one letter substitution is allowed. Here's what happens:

```
>>> pat.match("spam")       # Exact match
<_regex.Match object at 0x100547850>
>>> pat.match("slam")       # 1-letter substituted (match)
<_regex.Match object at 0x1005478b8>
>>> pat.match("slum")       # 2-letters substituted (no match)
>>>
```

There are additional options to specify insertions and deletions. For example, here is a pattern than allows, at most, one deletion, one substitution, and one insertion:

```
>>> pat = regex.compile("(?:spam){s<=1,d<=1,i<=1}$")
>>> pat.match("spm")      # Matches. 1 deletion
<_regex.Match object at 0x1005478b8>
>>> pat.match("sm")       # No match. 2 deletions
>>> pat.match("spaam")    # Match. 1 insertion
<_regex.Match object at 0x100547850>
>>> pat.match("slamm")    # Match. 1 insertion, 1 substitution
<_regex.Match object at 0x1005478b8>
>>> pat.match("slum")     # Match. 1 deletion, 1 insertion, 1 substition
```

Each insertion, substition, or deletion is counted separately and can be combined to match a wide range of words. If you wanted to narrow it down, you could just put a limit on the number of combined errors. For example:

```
>>> pat = regex.compile("(?:spam){s,i,d,e<=1}")
>>> pat2.match("spam")      # Match exact
<_regex.Match object at 0x100547850>
>>> pat2.match("spm")       # Match, 1 deletion
<_regex.Match object at 0x1005478b8>
>>> pat2.match("spaam")     # Match, 1 insertion
<_regex.Match object at 0x100547850>
>>> pat2.match("slum")      # No match
>>>
```

Needless to say, fuzzy matching opens up new areas of possible application to regular expression matching.

Putting It All Together

As a final example, it is now possible to present a short script that tries to identify people drunk-tweeting from the city of Chicago:

```
# drunktweet.py
'''
Print out possible drunk tweets from the city of Chicago.
'''

import regex
import requests
import json

# Terms for being "wasted"
terms = { 'drunk','wasted','buzzed','hammered','plastered' }

# A fuzzy regex for people who can't type
pat = regex.compile(r"(?:\L<terms>){i,d,s,e<=1}$", regex.I, terms=terms)

# Connect to the Twitter streaming API
url   = "https://stream.twitter.com/1/statuses/filter.json"
parms = {
   'locations' : "-87.94,41.644,-87.523,42.023"   # Chicago
   }
auth  = ('username','password')
r = requests.post(url, data=parms, auth=auth)

# Print possible candidates
for line in r.iter_lines():
   if line:
      tweet = json.loads(line)
      status = tweet.get('text',u'')
      words = status.split()
      if any(pat.match(word) for word in words):
        print tweet['user']['screen_name'], status
```

Final Words

Although this article has only focused on two modules, there are many other efforts to improve upon the standard library (too many to list). In a future issue, I hope to discuss alternatives to some of the system libraries—especially use of the subprocess module. Stay tuned.