feedcache Documentation

Release 1.4.1

Doug Hellmann

March 13, 2013

CONTENTS

The feedcache package implements a class to wrap Mark Pilgrim's Universal Feed Parser module so that parameters can be used to cache the feed results locally instead of fetching the feed every time it is requested. Uses both etag and modified times for caching. The cache is parameterized to use different backend storage options.

Contents:

CHAPTER

ONE

INTRODUCTION

1.1 Cache

The `Cache` class manages feed data, fetching updates when the local contents have expired. It uses feedparser to retrieve the data, and supports both `ETag` and `If-Modified-Since` headers.

1.2 Storage API

Each `Cache` uses a storage object to hold on to the data. The storage object must support the dictionary API. Keys are the URLs of the feed. The data stored includes, but is not limited to, the parsed result of the feed.

1.3 Cache + shelve

Using `shelve` by itself works in a simple single-threaded case but it isn't clear from its documentation whether shelve supports write access from multiple concurrent threads. To ensure the shelf is not corrupted, a thread lock should be used. `CacheStorageLock` is a simple wrapper around shelve that uses a lock to prevent more than one thread from accessing the shelf simultaneously.

See `feedcache/cachestoragelock.py` and `feedcache/example.py` for more details.

1.4 Cache + shove

Using `CacheStorageLock` protects against corruption caused by multiple threads in the same process, but shelve still only allows one process to open a shelf file to write to it. In applications with multiple processes that need to modify the cache, the shove module, by L. C. Rees, is an excellent alternative. shove offers support for a variety of back-end storage options, including: relational databases, BSD-style databases, Amazon's S3 storage service, pickle files, and others.

See `feedcache/example_threads.py` for more details.

CHAPTER

TWO

CACHING RSS FEEDS WITH FEEDCACHE

Note: This article was originally printed in Python Magazine Volume 1 Issue 11, November, 2007.

The past several years have seen a steady increase in the use of RSS and Atom feeds for data sharing. Blogs, podcasts, social networking sites, search engines, and news services are just a few examples of data sources delivered via such feeds. Working with internet services requires care, because inefficiencies in one client implementation may cause performance problems with the service that can be felt by all of the consumers accessing the same server. In this article, I describe the development of the **feedcache** package, and give examples of how you can use it to optimize the use of data feeds in your application.

I frequently find myself wanting to listen to one or two episodes from a podcast, but not wanting to subscribe to the entire series. In order to scratch this itch, I built a web based tool, hosted at http://www.castsampler.com/, to let me pick and choose individual episodes from a variety of podcast feeds, then construct a single feed with the results. Now I subscribe to the single feed with my podcast client, and easily populate it with new episodes when I encounter any that sound interesting. The feedcache package was developed as part of this tool to manage accessing and updating the feeds efficiently, and has been released separately under the BSD license.

2.1 Example Feed Data

The two most common publicly implemented formats for syndicating web data are RSS (in one of a few versions) and Atom. Both formats have a similar structure. Each feed begins with basic information about the data source (title, link, description, etc.). The introductory information is followed by a series of "items", each of which represents a resource like a blog post, news article, or podcast. Each item, in turn, has a title, description, and other information like when it was written. It may also refer to one or more attachments, or enclosures.

Listing 1 shows a sample RSS 2.0 feed and Listing 2 shows a sample Atom feed. Each sample listing contains one item with a single podcast enclosure. Both formats are XML, and contain essentially the same data. They use slightly different tag names though, and podcast enclosures are handled differently between the two formats, which can make working with different feed formats more work in some environments. Fortunately, Python developers do not need to worry about the differences in the feed formats, thanks to the Universal Feed Parser.

2.1.1 Listing 1

```
<?xml version="1.0" encoding="utf-8"?>

<rss version="2.0">
```

```
  <channel>
    <title>Sample RSS 2.0 Feed</title>
    <link>http://www.example.com/rss.xml</link>
    <description>Sample feed using RSS 2.0 format.</description>
    <language>en-us</language>
    <item>
      <title>item title goes here</title>
      <link>http://www.example.com/items/1/</link>
      <description>description goes here</description>
      <author>authoremail@example.com (author goes here)</author>
      <pubDate>Sat, 4 Aug 2007 15:00:36 -0000</pubDate>
      <guid>http://www.example.com/items/1/</guid>
      <enclosure url="http://www.example.com/items/1/enclosure" length="100" type="audio/mpeg">
      </enclosure>
    </item>
  </channel>
</rss>
```

2.1.2 Listing 2

```
<?xml version="1.0" encoding="utf-8"?>

<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en-us">
  <title>Sample Atom Feed</title>
  <link href="http://www.example.com/" rel="alternate"></link>
  <link href="http://www.example.com/atom.xml" rel="self"></link>
  <id>http://www.example.com/atom.xml</id>
  <updated>2007-08-04T15:00:36Z</updated>
  <entry>
    <title>title goes here</title>
    <link href="http://www.example.com/items/1/" rel="alternate"></link>
    <updated>2007-08-04T15:00:36Z</updated>
    <author>
      <name>author goes here</name>
      <email>authoremail@example.com</email>
    </author>
    <id>http://www.example.com/items/1/</id>
    <summary type="html">description goes here</summary>
    <link length="100" href="http://www.example.com/items/1/enclosure" type="audio/mpeg" rel="enclosu
    </link>
  </entry>
</feed>
```

2.2 Universal Feed Parser

Mark Pilgrim's Universal Feed Parser is an open source module that manages most aspects of downloading and parsing RSS and Atom feeds. Once the feed has been downloaded and parsed, the parser returns an object with all of the parsed data easily accessible through a single API, regardless of the original feed format.

Listing 3 shows a simple example program for accessing feeds with feedparser. On line 9, a URL from the command line arguments is passed to `feedparser.parse()` to be downloaded and parsed. The results are returned as a FeedParserDict. The properties of the FeedParserDict can be accessed via the dictionary API or using attribute names as illustrated on line 10.

2.2.1 Listing 3

```python
#!/usr/bin/env python
"""Print contents of feeds specified on the command line.
"""

import feedparser
import sys

for url in sys.argv[1:]:
    data = feedparser.parse(url)
    for entry in data.entries:
        print '%s: %s' % (data.feed.title, entry.title)
```

When the sample program in Listing 3 is run with the URL for the feed of **feedcache** project releases, it shows the titles for the releases available right now:

```
$ python Listing3.py http://feeds.feedburner.com/FeedcacheReleases
feedcache Releases: feedcache 0.1
feedcache Releases: feedcache 0.2
feedcache Releases: feedcache 0.3
feedcache Releases: feedcache 0.4
feedcache Releases: feedcache 0.5
```

Every time the program runs, it fetches the entire feed, whether the contents have changed or not. That inefficiency might not matter a great deal for a client program that is not run frequently, but the inefficiencies add up on the server when many clients access the same feed, especially if they check the feed on a regular basis. This inefficient behavior can become an especially bad problem for the server if the feed contents are produced dynamically, since each client incurs a certain amount of CPU, I/O, and bandwidth load needed to produce the XML representation of the feed. Some sites are understandably strict about how often a client can retrieve feeds, to cut down on heavy bandwidth and CPU consumers. Slashdot, for example, returns a special feed with a warning to any client that accesses their RSS feed too frequently over a short span of time.

2.3 A Different Type of Podcast Aggregator

A typical aggregator design would include a monitor to regularly download the feeds and store the fresh information about the feed and its contents in a database. The requirements for CastSampler are a little different, though.

CastSampler remembers the feeds to which a user has subscribed, but unlike other feed aggregators, it only downloads the episode metadata while the user is choosing episodes to add to their download feed. Since the user does not automatically receive every episode of every feed, the aggregator does not need to constantly monitor all of the feeds. Instead, it shows a list of episodes for a selected feed, and lets the user choose which episodes to download. Then it needs to remember those selected episodes later so it can produce the combined feed for the user's podcast client.

If every item from every feed was stored in the database, most of the data in the database would be for items that were never selected for download. There would need to be a way to remove old data from the database when it expired or was no longer valid, adding to the maintenance work for the site. Instead, CastSampler only uses the database to store information about episodes selected by the user. The rest of the data about the feed is stored outside of the database in a form that makes it easier to discard old data when the feed is updated. This division eliminates a lot of the data management effort behind running the site.

2.4 Feedcache Requirements

An important goal for this project was to make CastSampler a polite consumer of feeds, and ensure that it did not overload servers while a user was selecting podcast episodes interactively. By caching the feed data for a short period of time, CastSampler could avoid accessing feeds every time it needed to show the feed data. A persistent cache, written to disk, would let the data be reused even if the application was restarted, such as might happen during development. Using a cache would also improve responsiveness, since reading data from the local disk would be faster than fetching the feed from the remote server. To further reduce server load, **feedcache** is designed to take advantage of conditional GET features of HTTP, to avoid downloading the full feed whenever possible.

Another goal was to have a small API for the cache. It should take care of everything for the caller, so there would not need to be many functions to interact with it. To retrieve the contents of a feed, the caller should only have to provide the URL for that feed. All other information needed to track the freshness of the data in the cache would be managed internally.

It was also important for the cache to be able to store data in multiple ways, to make it more flexible for other programmers who might want to use it. Although CastSampler was going to store the cache on disk, other applications with more computing resources or tighter performance requirements might prefer to hold the cache in memory. Using disk storage should not be hard coded into the cache management logic.

These requirements led to a design which split the responsibility for managing the cached data between two objects. The **Cache** object tracks information about a feed so it can download the latest version as efficiently as possible, only when needed. Persistent storage of the data in the cache is handled by a separate back end storage object. Dividing the responsibilities in this way maximizes the flexibility of the **Cache**, since it can concentrate on tracking whether the feed is up to date without worrying about storage management. It also let **Cache** users take advantage of multiple storage implementations.

2.5 The Cache Class

Once the basic requirements and a skeleton design were worked out, the next step was to start writing tests so the implementation of **Cache** could begin. Working with a few simple tests would clarify how a **Cache** user would want to access feeds. The first test was to verify that the **Cache** would fetch feed data.

```
import unittest, cache
class CacheTest(unittest.TestCase):
    def testFetch(self):
        c = cache.Cache({})
        parsed_feed = c.fetch('http://feeds.feedburner.com/FeedcacheReleases')
        self.failUnless(parsed_feed.entries)
```

Since the design separated storage and feed management responsibilities, it was natural to pass the storage handler to the **Cache** when it is initialized. The dictionary API is used for the storage because there are several storage options available that support it. The shelve module in the Python standard library stores data persistently using an object that conforms to the dictionary API, as does the shove library from L.C. Rees. Either library would work well for the final application. For initial testing, using a simple dictionary to hold the data in memory was convenient, since that meant the tests would not need any external resources.

After constructing the **Cache**, the next step in the test is to retrieve a feed. I considered using using the `__getitem__()` hook, but since **Cache** would not support any of the other dictionary methods, I rejected it in favor of an explicit method, `fetch()`. The caller passes a feed URL to `fetch()`, which returns a FeedParserDict instance. Listing 4 shows the first version of the **Cache** class that works for the test as it is written. No actual caching is being done, yet. The **Cache** instance simply uses the **feedparser** module to retrieve and parse the feed.

2.5.1 Listing 4

```python
#!/usr/bin/env python
"""The first version of Cache
"""

import unittest
import feedparser

class Cache:
    def __init__(self, storage):
        self.storage = storage
        return

    def fetch(self, url):
        return feedparser.parse(url)

class CacheTest(unittest.TestCase):

    def testFetch(self):
        c = Cache({})
        parsed_feed = c.fetch('http://feeds.feedburner.com/FeedcacheReleases')
        self.failUnless(parsed_feed.entries)
        return

if __name__ == '__main__':
    unittest.main()
```

2.6 Throttling Downloads

Now that **Cache** could successfully download feed data, the first optimization to make was to hold on to the data and track its age. Then for every call to `fetch()`, **Cache** could first check to see if fresh data was already available locally before going out to the server to download the feed again.

Listing 5 shows the version of **Cache** with a download throttle, in the form of a `timeToLiveSeconds` parameter. Items already in the cache will be reused until they are older than `timeToLiveSeconds`. The default value for `timeToLiveSeconds` means that any given feed will not be checked more often than every five minutes.

2.6.1 Listing 5

```python
#!/usr/bin/env python
"""The first version of Cache
"""

import time
import unittest
import feedparser

class Cache:
    def __init__(self, storage, timeToLiveSeconds=300):
        self.storage = storage
        self.time_to_live = timeToLiveSeconds
        return

```

```
15      def fetch(self, url):
16          now = time.time()
17          cached_time, cached_content = self.storage.get(url, (None, None))
18
19          # Does the storage contain a version of the data
20          # which is older than the time-to-live?
21          if cached_time is not None:
22              age = now - cached_time
23              if age <= self.time_to_live:
24                  return cached_content
25
26          parsed_data = feedparser.parse(url)
27          self.storage[url] = (now, parsed_data)
28          return parsed_data
29
30  class CacheTest(unittest.TestCase):
31
32      def testFetch(self):
33          c = Cache({})
34          parsed_feed = c.fetch('http://feeds.feedburner.com/FeedcacheReleases')
35          self.failUnless(parsed_feed.entries)
36          return
37
38      def testReuseContentsWithinTimeToLiveWindow(self):
39          url = 'http://feeds.feedburner.com/FeedcacheReleases'
40          c = Cache({ url:(time.time(), 'prepopulated cache')})
41          cache_contents = c.fetch(url)
42          self.failUnlessEqual(cache_contents, 'prepopulated cache')
43          return
44
45  if __name__ == '__main__':
46      unittest.main()
```

The new implementation of `fetch()` stores the current time along with the feed data when the storage is updated. When `fetch()` is called again with the same URL, the time in the cache is checked against the current time to determine if the value in the cache is fresh enough. The test on line 38 verifies this behavior by pre-populating the **Cache**'s storage with data, and checking to see that the existing cache contents are returned instead of the contents of the feed.

2.7 Conditional HTTP GET

Conditional HTTP GET allows a client to tell a server something about the version of a feed the client already has. The server can decide if the contents of the feed have changed and, if they have not, send a short status code in the HTTP response instead of a complete copy of the feed data. Conditional GET is primarily a way to conserve bandwidth, but if the feed has not changed and the server's version checking algorithm is efficient then the server may use fewer CPU resources to prepare the response, as well.

When a server implements conditional GET, it uses extra headers with each response to notify the client. There are two headers involved, and the server can use either or both together, in case the client only supports one. **Cache** supports both headers.

Although timestamps are an imprecise way to detect change, since the time on different servers in a pool might vary slightly, they are simple to work with. The `Last-Modified` header contains a timestamp value that indicates when the feed contents last changed. The client sends the timestamp back to the server in the next request as `If-Modified-Since`. The server then compares the dates to determine if the feed has been modified since the last request from the client.

A more precise way to determine if the feed has changed is to use an *Entity Tag* in the `ETag` header. An `ETag` is a hashed representation of the feed state, or of a value the server can use to quickly determine if the feed has been updated. The data and algorithm for computing the hash is left up to the server, but it should be less expensive than returning the feed contents or there won't be any performance gains. When the client sees an `ETag` header, it can send the associated value back to the server with the next request in the `If-None-Match` request header. When the server sees `If-None-Match`, it computes the current hash and compares it to the value sent by the client. If they match, the feed has not changed.

When using either `ETag` or modification timestamps, if the server determines that the feed has not been updated since the previous request, it returns a response code of `304`, or "Not Modified" and includes nothing in the body of the response. When it sees the `304` status in the response from the server, the client should reuse the version of the feed it already has.

2.8 Creating a Test Server

In order to write correct tests to exercise conditional GET in **feedcache**, more control over the server would be important. The feedburner URL used in the earlier tests might be down, or return different data if a feed was updated. It would be necessary for the server to respond reliably with data the test code knew in advance, and to be sure it would not stop responding if it was queried too often by the tests. The tests also control which of the headers (`ETag` or `If-Modified-Since`) was used to determine if the feed had changed, so both methods could be tested independently. The solution was to write a small test HTTP server that could be managed by the unit tests and configured as needed. Creating the test server was easy, using a few standard library modules.

The test server code, along with a base class for unit tests that use it, can be found in Listing 6. The `TestHTTPServer` (line 91) is derived from `BaseHTTPServer.HTTPServer`. The `serve_forever()` method (line 112) has been overridden with an implementation that checks a flag after each request to see if the server should keep running. The test harness sets the flag to stop the test server after each test. The `serve_forever()` loop also counts the requests successfully processed, so the tests can determine how many times the **Cache** fetches a feed.

2.8.1 Listing 6

```
1  #!/usr/bin/env python
2  """Simple HTTP server for testing the feed cache.
3  """
4
5  import BaseHTTPServer
6  import email.utils
7  import logging
8  import md5
9  import threading
10 import time
11 import unittest
12 import urllib
13
14
15 def make_etag(data):
16     """Given a string containing data to be returned to the client,
17     compute an ETag value for the data.
18     """
19     _md5 = md5.new()
20     _md5.update(data)
21     return _md5.hexdigest()
22
```

```

class TestHTTPHandler(BaseHTTPServer.BaseHTTPRequestHandler):
    "HTTP request handler which serves the same feed data every time."

    FEED_DATA = """<?xml version="1.0" encoding="utf-8"?>

<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en-us">
  <title>CacheTest test data</title>
  <link href="http://localhost/feedcache/" rel="alternate"></link>
  <link href="http://localhost/feedcache/atom/" rel="self"></link>
  <id>http://localhost/feedcache/</id>
  <updated>2006-10-14T11:00:36Z</updated>
  <entry>
    <title>single test entry</title>
    <link href="http://www.example.com/" rel="alternate"></link>
    <updated>2006-10-14T11:00:36Z</updated>
    <author>
      <name>author goes here</name>
      <email>authoremail@example.com</email>
    </author>
    <id>http://www.example.com/</id>
    <summary type="html">description goes here</summary>
    <link length="100" href="http://www.example.com/enclosure" type="text/html" rel="enclosure">
    </link>
  </entry>
</feed>"""

    # The data does not change, so save the ETag and modified times
    # as class attributes.
    ETAG = make_etag(FEED_DATA)
    MODIFIED_TIME = email.utils.formatdate(usegmt=True)

    def do_GET(self):
        "Handle GET requests."

        if self.path == '/shutdown':
            # Shortcut to handle stopping the server
            self.server.stop()
            self.send_response(200)

        else:
            incoming_etag = self.headers.get('If-None-Match', None)
            incoming_modified = self.headers.get('If-Modified-Since', None)

            send_data = True

            # Does the client have the same version of the data we have?
            if self.server.apply_modified_headers:
                if incoming_etag == self.ETAG:
                    self.send_response(304)
                    send_data = False

                elif incoming_modified == self.MODIFIED_TIME:
                    self.send_response(304)
                    send_data = False

            # Now optionally send the data, if the client needs it
            if send_data:
                self.send_response(200)
                self.send_header('Content-Type', 'application/atom+xml')
                self.send_header('ETag', self.ETAG)
                self.send_header('Last-Modified', self.MODIFIED_TIME)
                self.end_headers()

                self.wfile.write(self.FEED_DATA)
        return


class TestHTTPServer(BaseHTTPServer.HTTPServer):
    """HTTP Server which counts the number of requests made
    and can stop based on client instructions.
    """

    def __init__(self, applyModifiedHeaders=True):
        self.apply_modified_headers = applyModifiedHeaders
        self.keep_serving = True
        self.request_count = 0
        BaseHTTPServer.HTTPServer.__init__(self, ('', 9999), TestHTTPHandler)
        return

    def getNumRequests(self):
        "Return the number of requests which have been made on the server."
        return self.request_count

    def stop(self):
        "Stop serving requests, after the next request."
        self.keep_serving = False
        return

    def serve_forever(self):
        "Main loop for server"
        while self.keep_serving:
            self.handle_request()
            self.request_count += 1
        return


class HTTPTestBase(unittest.TestCase):
    "Base class for tests that use a TestHTTPServer"

    TEST_URL = 'http://localhost:9999/'

    CACHE_TTL = 0

    def setUp(self):
        self.server = self.getServer()
        self.server_thread = threading.Thread(target=self.server.serve_forever)
        self.server_thread.setDaemon(True) # so the tests don't hang if cleanup fails
        self.server_thread.start()
        return

    def getServer(self):
        "Return a web server for the test."
        return TestHTTPServer()

    def tearDown(self):
```

```
139         # Stop the server thread
140         ignore = urllib.urlretrieve('http://localhost:9999/shutdown')
141         time.sleep(1)
142         self.server.server_close()
143         self.server_thread.join()
144         return
145 
146 
147 class HTTPTest(HTTPTestBase):
148 
149     def testResponse(self):
150         # Verify that the server thread responds
151         # without error.
152         filename, response = urllib.urlretrieve(self.TEST_URL)
153         return
154 
155 if __name__ == '__main__':
156     unittest.main()
```

The test server processes incoming HTTP requests with `TestHTTPHandler` (line 24), derived from `BaseHTTPServer.BaseHTTPRequestHandler`. `TestHTTPHandler` implements `do_GET()` (line 55) to respond to HTTP GET requests. Feed data for the tests is hard coded in the `FEED_DATA` class attribute (line 27). The URL path `/shutdown` is used to tell the server to stop responding to requests. All other paths are treated as requests for the feed data. The requests are processed by checking the `If-None-Match` and `If-Modified-Since` headers, and responding either with a `304` status or with the static feed data.

`HTTPTestBase` is a convenience base class to be used by other tests. It manages a `TestHTTPServer` instance in a separate thread, so the tests can all run in a single process. Listing 7 shows what the existing tests look like, rewritten to use the `HTTPTestBase` as a base class. The only differences are the base class for the tests and the use of `self.TEST_URL`, which points to the local test server instead of the **feedburner** URL from Listing 5.

2.8.2 Listing 7

```
#!/usr/bin/env python
"""The first version of Cache
"""

import time
import unittest
import feedparser
from Listing5 import Cache
from Listing6 import HTTPTestBase

class CacheTest(HTTPTestBase):

    def testFetch(self):
        c = Cache({})
        parsed_feed = c.fetch(self.TEST_URL)
        self.failUnless(parsed_feed.entries)
        return

    def testReuseContentsWithinTimeToLiveWindow(self):
        c = Cache({ self.TEST_URL:(time.time(), 'prepopulated cache')})
        cache_contents = c.fetch(self.TEST_URL)
        self.failUnlessEqual(cache_contents, 'prepopulated cache')
        return

```

```
if __name__ == '__main__':
    unittest.main()
```

2.9 Implementing Conditional HTTP GET

With these testing tools in place, the next step was to enhance the **Cache** class to monitor and use the conditional HTTP GET parameters. Listing 8 shows the final version of **Cache** with these features. The `fetch()` method has been enhanced to send the `ETag` and modified time from the cached version of the feed to the server, when they are available.

2.9.1 Listing 8

```
#!/usr/bin/env python
"""Cache class with conditional HTTP GET support.
"""

import feedparser
import time
import unittest
import UserDict

import Listing6 # For the test base class

class Cache:

    def __init__(self, storage, timeToLiveSeconds=300, userAgent='feedcache'):
        self.storage = storage
        self.time_to_live = timeToLiveSeconds
        self.user_agent = userAgent
        return

    def fetch(self, url):
        modified = None
        etag = None
        now = time.time()

        cached_time, cached_content = self.storage.get(url, (None, None))

        # Does the storage contain a version of the data
        # which is older than the time-to-live?
        if cached_time is not None:
            if self.time_to_live:
                age = now - cached_time
                if age <= self.time_to_live:
                    return cached_content

            # The cache is out of date, but we have
            # something.  Try to use the etag and modified_time
            # values from the cached content.
            etag = cached_content.get('etag')
            modified = cached_content.get('modified')

        # We know we need to fetch, so go ahead and do it.
        parsed_result = feedparser.parse(url,
                                             agent=self.user_agent,
                                             modified=modified,
                                             etag=etag,
                                             )

        status = parsed_result.get('status', None)
        if status == 304:
            # No new data, based on the etag or modified values.
            # We need to update the modified time in the
            # storage, though, so we know that what we have
            # stored is up to date.
            self.storage[url] = (now, cached_content)

            # Return the data from the cache, since
            # the parsed data will be empty.
            parsed_result = cached_content
        elif status == 200:
            # There is new content, so store it unless there was an error.
            error = parsed_result.get('bozo_exception')
            if not error:
                self.storage[url] = (now, parsed_result)

        return parsed_result


class SingleWriteMemoryStorage(UserDict.UserDict):
    """Cache storage which only allows the cache value
    for a URL to be updated one time.
    """

    def __setitem__(self, url, data):
        if url in self.keys():
            modified, existing = self[url]
            # Allow the modified time to change,
            # but not the feed content.
            if data[1] != existing:
                raise AssertionError('Trying to update cache for %s to %s' \
                                         % (url, data))
        UserDict.UserDict.__setitem__(self, url, data)
        return


class CacheConditionalGETTest(Listing6.HTTPTestBase):

    def setUp(self):
        Listing6.HTTPTestBase.setUp(self)
        self.cache = Cache(storage=SingleWriteMemoryStorage(),
                           timeToLiveSeconds=0, # so we do not reuse the local copy
                           )
        return

    def testFetchOnceForEtag(self):
        # Fetch data which has a valid ETag value, and verify
        # that while we hit the server twice the response
        # codes cause us to use the same data.

        # First fetch populates the cache
        response1 = self.cache.fetch(self.TEST_URL)
        self.failUnlessEqual(response1.feed.title, 'CacheTest test data')

        # Remove the modified setting from the cache so we know
        # the next time we check the etag will be used
        # to check for updates.  Since we are using an in-memory
        # cache, modifying response1 updates the cache storage
        # directly.
        response1['modified'] = None

        # Wait so the cache data times out
        time.sleep(1)

        # This should result in a 304 status, and no data from
        # the server.  That means the cache won't try to
        # update the storage, so our SingleWriteMemoryStorage
        # should not raise and we should have the same
        # response object.
        response2 = self.cache.fetch(self.TEST_URL)
        self.failUnless(response1 is response2)

        # Should have hit the server twice
        self.failUnlessEqual(self.server.getNumRequests(), 2)
        return

    def testFetchOnceForModifiedTime(self):
        # Fetch data which has a valid Last-Modified value, and verify
        # that while we hit the server twice the response
        # codes cause us to use the same data.

        # First fetch populates the cache
        response1 = self.cache.fetch(self.TEST_URL)
        self.failUnlessEqual(response1.feed.title, 'CacheTest test data')

        # Remove the etag setting from the cache so we know
        # the next time we check the modified time will be used
        # to check for updates.  Since we are using an in-memory
        # cache, modifying response1 updates the cache storage
        # directly.
        response1['etag'] = None

        # Wait so the cache data times out
        time.sleep(1)

        # This should result in a 304 status, and no data from
        # the server.  That means the cache won't try to
        # update the storage, so our SingleWriteMemoryStorage
        # should not raise and we should have the same
        # response object.
        response2 = self.cache.fetch(self.TEST_URL)
        self.failUnless(response1 is response2)

        # Should have hit the server twice
        self.failUnlessEqual(self.server.getNumRequests(), 2)
        return

if __name__ == '__main__':
    unittest.main()
```

The **FeedParserDict** object returned from `feedparser.fetch()` conveniently includes the `ETag` and modified timestamp, if the server sent them. On lines 38-39, once the cached feed is determined to be out of date, the `ETag` and modified values are retrieved so they can be passed in to `feedparser.parse()` on line 42.

Since the updated client sends `ETag` and `If-Modified-Since` headers, the server may now respond with a status code indicating that the cached copy of the data is still valid. It is no longer sufficient to simply store the response from the server before returning it. The status code must be checked, as on line 49, and if the status is `304` then the timestamp of the cached copy is updated. If the timestamp was not updated, then as soon as the cached copy of the feed exceeded the time-to-live, the **Cache** would request a new copy of the feed from the server every time the feed was accessed. Updating the timestamp ensures that the download throttling remains enforced.

Separate tests for each conditional GET header are implemented in `CacheConditionalGETTest`. To verify that the **Cache** handles the `304` status code properly and does not try to update the contents of the storage on a second fetch, these tests use a special storage class. The **SingleWriteMemoryStorage** raises an **AssertionError** if the a value is modified after it is set the first time. An `AssertionError` is used, because that is how `unittest.TestCase` signals a test failure, and modifying the contents of the storage is a failure for these tests.

Each test method of `CacheConditionalGETTest` verifies handling for one of the conditional GET headers at a time. Since the test server always sets both headers, each test clears one value from the cache before making the second request. The remaining header value is sent to the server as part of the second request, and the server responds with the `304` status code.

2.10 Persistent Storage With shelve

All of the examples and tests so far have used in-memory storage options. For CastSampler, though, the cache of feed data needed to be stored on disk. As mentioned earlier, the **shelve** module in the standard library provides a simple persistent storage mechanism. It also conforms to the dictionary API used by the **Cache** class.

Using **shelve** by itself works in a simple single threaded case but it isn't clear from its documentation whether **shelve** supports write access from multiple concurrent threads. To ensure the shelf is not corrupted, a thread lock should be used. `CacheStorageLock` is a simple wrapper around **shelve** that uses a lock to prevent more than one thread from accessing the shelf simultaneously. Listing 9 contains the code for the `CacheStorageLock` and a test that illustrates using it to combine a **Cache** and **shelve**.

2.10.1 Listing 9

```python
#!/usr/bin/env python

from __future__ import with_statement

"""Using Cache with shelve.
"""

import os
import shelve
import tempfile
import threading
import unittest

from Listing6 import HTTPTestBase
from Listing8 import Cache

class CacheStorageLock:

    def __init__(self, shelf):
        self.lock = threading.Lock()
        self.shelf = shelf
        return

    def __getitem__(self, key):
        with self.lock:
            return self.shelf[key]

    def get(self, key, default=None):
        with self.lock:
            try:
                return self.shelf[key]
            except KeyError:
                return default

    def __setitem__(self, key, value):
        with self.lock:
            self.shelf[key] = value


class CacheShelveTest(HTTPTestBase):

    def setUp(self):
        HTTPTestBase.setUp(self)
        handle, self.shelve_filename = tempfile.mkstemp('.shelve')
        os.close(handle) # we just want the file name, so close the open handle
        os.unlink(self.shelve_filename) # remove empty file so shelve is not confused
        return

    def tearDown(self):
        try:
            os.unlink(self.shelve_filename)
        except AttributeError:
            pass
        HTTPTestBase.tearDown(self)
        return

    def test(self):
        storage = shelve.open(self.shelve_filename)
        locking_storage = CacheStorageLock(storage)
        try:
            fc = Cache(locking_storage)

            # First fetch the data through the cache
            parsed_data = fc.fetch(self.TEST_URL)
            self.failUnlessEqual(parsed_data.feed.title, 'CacheTest test data')

            # Now retrieve the same data directly from the shelf
            modified, shelved_data = storage[self.TEST_URL]

            # The data should be the same
            self.failUnlessEqual(parsed_data, shelved_data)
        finally:
            storage.close()
        return


if __name__ == '__main__':
    unittest.main()
```

The test `setUp()` method uses `tempfile` to create a temporary filename for the cache. The temporary file has to be deleted in `setUp()` because if the file exists, but is empty, **shelve** cannot determine which database module to use to open an empty file. The `test()` method fetches the data from the server, then compares the returned data with the data in the shelf to verify that they are the same.

`CacheStorageLock` uses a `threading.Lock` instance to control access to the shelf. It only manages access for the methods known to be used by **Cache**. The lock is acquired and released using the `with` statement, which is new for Python 2.6. Since this code was written with Python 2.5, the module starts with a `from __future__ import` statement to enable the syntax for `with`.

2.11 Other Persistence Options

At any one time, **shelve** only allows one process to open a shelf file to write to it. In applications with multiple processes that need to modify the cache, alternative storage options are desirable. **Cache** treats its storage object as a dictionary, so any class that conforms to the dictionary API can be used for back end storage. The `shove` module, by L. C. Rees, uses the dictionary API and offers support for a variety of back end storage options. The supported options include relational databases, BSD-style databases, Amazon's S3 storage service, and others.

The filesystem store option was particularly interesting for CastSampler. With **shove**'s file store, each key is mapped to a filename. The data associated with the key is pickled and stored in the file. By using separate files, it is possible to have separate threads and processes updating the cache simultaneously. Although the **shove** file implementation doesn't handle file locking, for my purposes it was unlikely that two threads would try to update the same feed at the same time.

Listing 10 includes a test that illustrates using **shove** file storage with **feedcache**. The primary difference in the APIs for **shove** and **shelve** is the syntax for specifying the storage destination. Shove uses a URL syntax to indicate which back end should be used. The format for each back end is described in the docstrings.

2.11.1 Listing 10

```
#!/usr/bin/env python
"""Tests with shove filesystem storage.
"""

import os
import shove
import tempfile
import threading
import unittest

from Listing6 import HTTPTestBase
from Listing8 import Cache

class CacheShoveTest(HTTPTestBase):

    def setUp(self):
        HTTPTestBase.setUp(self)
        self.shove_dirname = tempfile.mkdtemp('shove')
        return

    def tearDown(self):
        try:
            os.system('rm -rf %s' % self.storage_dirname)
        except AttributeError:
            pass
        HTTPTestBase.tearDown(self)
        return

    def test(self):
        # First fetch the data through the cache
        storage = shove.Shove('file://' + self.shove_dirname)
        try:
            fc = Cache(storage)
            parsed_data = fc.fetch(self.TEST_URL)
            self.failUnlessEqual(parsed_data.feed.title, 'CacheTest test data')
        finally:
            storage.close()

        # Now retrieve the same data directly from the shelf
        storage = shove.Shove('file://' + self.shove_dirname)
        try:
            modified, shelved_data = storage[self.TEST_URL]
        finally:
            storage.close()

        # The data should be the same
        self.failUnlessEqual(parsed_data, shelved_data)
        return


if __name__ == '__main__':
    unittest.main()
```

2.12 Using feedcache With Multiple Threads

Up to this point, all of the examples have been running in a single thread driven by the **unittest** framework. Now that integrating **shove** and **feedcache** has been shown to work, it is possible to take a closer look at using multiple threads to fetch feeds, and build a more complex example application. Spreading the work of fetching data into multiple processing threads is more complicated, but yields better performance under most circumstances because while one thread is blocked waiting for data from the network, another thread can take over and process a different URL.

Listing 11 shows a sample application which accepts URLs as arguments on the command line and prints the titles of all of the entries in the feeds. The results may be mixed together, depending on how the processing control switches between active threads. This example program is more like a traditional feed aggregator, since it processes every entry of every feed.

2.12.1 Listing 11

```
#!/usr/bin/env python
"""Example use of feedcache.Cache combined with threads.
"""

import Queue
import sys
import shove
import threading
```

```

from Listing8 import Cache

MAX_THREADS=5
OUTPUT_DIR='/tmp/feedcache_example'


def main(urls=[]):

    if not urls:
        print 'Specify the URLs to a few RSS or Atom feeds on the command line.'
        return

    # Add the URLs to a queue
    url_queue = Queue.Queue()
    for url in urls:
        url_queue.put(url)

    # Add poison pills to the url queue to cause
    # the worker threads to break out of their loops
    for i in range(MAX_THREADS):
        url_queue.put(None)

    # Track the entries in the feeds being fetched
    entry_queue = Queue.Queue()

    print 'Saving feed data to', OUTPUT_DIR
    storage = shove.Shove('file://' + OUTPUT_DIR)
    try:

        # Start a few worker threads
        worker_threads = []
        for i in range(MAX_THREADS):
            t = threading.Thread(target=fetch_urls,
                                 args=(storage, url_queue, entry_queue,))
            worker_threads.append(t)
            t.setDaemon(True)
            t.start()

        # Start a thread to print the results
        printer_thread = threading.Thread(target=print_entries, args=(entry_queue,))
        printer_thread.setDaemon(True)
        printer_thread.start()

        # Wait for all of the URLs to be processed
        url_queue.join()

        # Wait for the worker threads to finish
        for t in worker_threads:
            t.join()

        # Poison the print thread and wait for it to exit
        entry_queue.put((None,None))
        entry_queue.join()
        printer_thread.join()

    finally:
        storage.close()
```

```
67      return
68
69
70  def fetch_urls(storage, input_queue, output_queue):
71      """Thread target for fetching feed data.
72      """
73      c = Cache(storage)
74
75      while True:
76          next_url = input_queue.get()
77          if next_url is None: # None causes thread to exit
78              input_queue.task_done()
79              break
80
81          feed_data = c.fetch(next_url)
82          for entry in feed_data.entries:
83              output_queue.put( (feed_data.feed, entry) )
84          input_queue.task_done()
85      return
86
87
88  def print_entries(input_queue):
89      """Thread target for printing the contents of the feeds.
90      """
91      while True:
92          feed, entry = input_queue.get()
93          if feed is None: # None causes thread to exist
94              input_queue.task_done()
95              break
96
97          print '%s: %s' % (feed.title, entry.title)
98          input_queue.task_done()
99      return
100
101
102 if __name__ == '__main__':
103     main(sys.argv[1:])
```

The design uses queues to pass data between two different types of threads to work on the feeds. Multiple threads use **feedcache** to fetch feed data. Each of these threads has its own **Cache**, but they all share a common **shove** store. A single thread waits for the feed entries to be added to its queue, and then prints each feed title and entry title.

The `main()` function sets up two different queues for passing data in and out of the worker threads. The `url_queue` (lines 23-25) contains the URLs for feeds, taken from the command line arguments. The `entry_queue` (line 33) is used to pass feed content from the threads that fetch the feeds to the queue that prints the results. A **shove** filesystem store (line 36) is used to cache the feeds. Once all of the worker threads are started (lines 40-51), the rest of the main program simply waits for each stage of the work to be completed by the threads.

The last entries added to the `url_queue` are `None` values, which trigger the worker thread to exit. When the `url_queue` has been drained (line 54), the worker threads can be cleaned up. After the worker threads have finished, `(None, None)` is added to the `entry_queue` to trigger the printing thread to exit when all of the entries have been printed.

The `fetch_urls()` function (lines 70-85) runs in the worker threads. It takes one feed URL at a time from the input queue, retrieves the feed contents from a cache, then adds the feed entries to the output queue. When the item taken out of the queue is `None` instead of a URL string, it is interpreted as a signal that the thread should break out of its processing loop. Each thread running `fetch_urls()` creates a local **Cache** instance using a common storage back end. Sharing the storage ensures that all of the feed data is written to the same place, while creating a local **Cache** instance ensures threads can fetch data in parallel.

The consumer of the queue of entries is `print_entries()` (lines 88-99). It takes one entry at a time from the queue and prints the feed and entry titles. Only one thread runs `print_entries()`, but a separate thread is used so that output can be produced as soon as possible, instead of waiting for all of the `fetch_urls()` threads to complete before printing the feed contents.

Running the program produces output similar to the example in Listing 3:

```
$ python Listing11.py http://feeds.feedburner.com/FeedcacheReleases
Saving feed data to /tmp/feedcache_example
feedcache Releases: feedcache 0.1
feedcache Releases: feedcache 0.2
feedcache Releases: feedcache 0.3
feedcache Releases: feedcache 0.4
feedcache Releases: feedcache 0.5
```

The difference is that it takes much less time to run the program in Listing 11 when multiple feeds are passed on the command line, and when some of the data has already been cached.

2.13 Future Work

The current version of **feedcache** meets most of the requirements for **CastSampler**, but there is still room to improve it as a general purpose tool. It would be nice if it offered finer control over the length of time data stays in the cache, for example. And, although **shove** is a completely separate project, **feedcache** would be more reliable if **shove**'s file storage were used file locking, to prevent corruption when two threads or processes write to the same part of the cache at the same time.

Determining how long to hold the data in a cache can be a tricky problem. With web content such as RSS and Atom feeds, the web server may offer hints by including explicit expiration dates or caching instructions. HTTP headers such as `Expires` and `Cache-Control` can include details beyond the `Last-Modified` and `ETag` values already being handled by the **Cache**. If the server uses additional cache headers, **feedparser** saves the associated values in the **FeedParserDict**. To support the caching hints, **feedcache** would need to be enhanced to understand the rules for the `Cache-Control` header, and to save the expiration time as well as the time-to-live for each feed.

Supporting a separate time-to-live value for each feed would let **feedcache** use a different refresh throttle for different sites. Data from relatively infrequently updated feeds, such as Slashdot, would stay in the cache longer than data from more frequently updated feeds, such as a Twitter feed. Applications that use **feedcache** in a more traditional way would be able to adjust the update throttle for each feed separately to balance the freshness of the data in the cache and the load placed on the server.

2.14 Conclusions

Original sources of RSS and Atom feeds are being created all the time as new and existing applications expose data for syndication. With the development of mashup tools such as Yahoo! Pipes and Google's Mashup Editor, these feeds can be combined, filtered, and expanded in new and interesting ways, creating even more sources of data. I hope this article illustrates how building your own applications to read and manipulate syndication feeds in Python with tools like feedparser and **feedcache** is easy, even while including features that make your program cooperate with servers to manage load.

I would like to offer a special thanks to Mrs. PyMOTW for her help editing this article.

CHAPTER

THREE

HISTORY

1.4.1

- Moved the documentation to readthedocs.org.
- Updated packaging to use distribute instead of distutils.

1.3.1

- Updated build to work with hg and migrated code to bitbucket hosting. No source changes.

1.3

- Supports purging cache contents, based on a suggestion by Thomas Perl.

1.2

- Add a bit of documentation.

1.1

  - New features based on a patch from Thomas Perl:
  - Unicode handling for URLs.
  - force_update flag
  - offline mode flag

1.0

- Lock down the API from the last alpha version, and move to beta status.

0.5

- Add tests that illustrate using `feedcache` with shove.

0.4

- API changes and additional tests.

0.3

- Eliminates a race condition in the backend storage API.

0.2

- Improve the ShelveStorage backend and add an example script to illustrate how to use the cache.

0.1

- Early alpha version with in-memory and shelve storage options for the backend.

CHAPTER

FOUR

LICENSE

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*