cookies Documentation

Release 1.0.0

Sasha Hart

March 02, 2017

Contents

1 What is this and what is it for? 1

2 Features 3

3 Things this is not meant to do 5

4 API Guide 7

4.1 Cookie objects 7

4.2 Cookies objects 9

4.3 Extension Mechanisms 11

CHAPTER 1

What is this and what is it for?

cookies.py is a Python module for working with HTTP cookies: parsing and rendering 'Cookie:' request headers and 'Set-Cookie:' response headers, and exposing a convenient API for creating and modifying cookies. It can be used as a replacement of Python's Cookie.py (aka http.cookies).

CHAPTER 2

Features

- Rendering according to the excellent new RFC 6265 (rather than using a unique ad hoc format inconsistently relating to unrealistic, very old RFCs which everyone ignored). Uses URL encoding to represent non-ASCII by default, like many other languages' libraries
- Liberal parsing, incorporating many complaints about Cookie.py barfing on common cookie formats which can be reliably parsed (e.g. search 'cookie' on the Python issue tracker)
- Well-documented code, with chapter and verse from RFCs (rather than arbitrary, undocumented decisions and huge tables of magic values, as you see in Cookie.py).
- Test coverage at 100%, with a much more comprehensive test suite than Cookie.py
- Single-source compatible with the following Python versions: 2.6, 2.7, 3.2, 3.3 and PyPy (2.7).
- Cleaner, less surprising API:

```
# old Cookie.py - this code is all directly from its docstring
>>> from Cookie import SmartCookie
>>> C = SmartCookie()
>>> # n.b. it's "smart" because it automatically pickles Python objects,
>>> # which is actually quite stupid for security reasons!
>>> C["rocky"] = "road"
>>> C["rocky"]["path"] = "/cookie"
>>> # So C["rocky"] is a string, except when it's a dict...
>>> # and why do I have to write [""] to access a fixed set of attrs?
>>> # Look at the atrocious way I render out a request header:
>>> C.output(attrs=[], header="Cookie:")
'Cookie: rocky=road'

# new cookies.py
>>> from cookies import Cookies, Cookie
>>> cookies = Cookies(rocky='road')
>>> # Can also write explicitly: cookies['rocky'] = Cookie['road']
>>> cookies['rocky'].path = "/cookie"
>>> cookies.render_request()
'rocky=road'
```

- Friendly to customization, extension, and reuse of its parts. Unlike Cookie.py, it doesn't lock all implementation inside its own classes (forcing you to write ugly wrappers as Django, Trac, Werkzeug/Flask, web.py and Tornado had to do). You can suppress minor parse exceptions with parameters rather than subclass wrappers. You can plug in your own parsers, renderers and validators for new or existing cookie attributes. You can render the data out in a dict. You can easily use the underlying imperative API or even lift the parser's regexps for your own parser or project. They are very well documented and relate directly to RFCs, so you know exactly what you are getting and why. It's MIT-licensed so do what you want (but I'd love to know what use you are getting from it!)

- One file, so you can just drop cookies.py into your project if you like
- MIT license, so you can use it in whatever you want with no strings

CHAPTER 3

Things this is not meant to do

While this is intended to be a good module for handling cookies, it does not even try to do any of the following:

- Maintain backward compatibility with Cookie.py, which would mean inheriting its confusions and bugs
- Implement RFCs 2109 or 2965, which have always been ignored by almost everyone and are now obsolete as well
- Handle every conceivable output from terrible legacy apps, which is not possible to do without lots of silent data loss and corruption (the parser does try to be liberal as possible otherwise, though)
- Provide a means to store pickled Python objects in cookie values (that's a big security hole)

This doesn't compete with the cookielib (http.cookiejar) module in the Python standard library, which is specifically for implementing cookie storage and similar behavior in an HTTP client such as a browser. Things cookielib does that this doesn't:

- Write to or read from browsers' cookie stores or other proprietary formats for storing cookie data in files
- Handle the browser/client logic like deciding which cookies to send or discard, etc.

If you are looking for a cookie library but neither this one nor cookielib will help, you might also consider the implementations in WebOb or Bottle.

CHAPTER 4

API Guide

Okay, so this is supposed to be a very nice module for parsing, manipulating and rendering HTTP cookie data. So how do you use this thing?

Two interfaces are exposed: a collection class named Cookies, and a class named Cookie to represent each particular name, value and set of attributes. If you want to, you can just ignore Cookie and just use Cookies as a dictionary of objects with name and value attributes.

Cookie objects

Each individual Cookie object can be queried and manipulated as a normal Python object with name and value attributes (and other attributes corresponding to cookie attributes). Normally this is all you'll need, but the following describes most of the available facilities.

A Cookie object can be created explicitly.

```
>>> from cookies import Cookie
>>> cookie = Cookie('a', 'b')
>>> cookie.name
'a'
>>> cookie.value
'b'
>>> cookie.value = 'f'
>>> cookie.value
'f'
```

You can also explicitly create a Cookie object with special attributes set in the constructor.

```
>>> cookie = Cookie('a', 'b', comment="need to track a")
>>> cookie.comment
'need to track a'
```

You can also make a single Cookie by parsing a string:

```
>>> cookie = Cookie.from_string('Set-Cookie: x=y')
>>> (cookie.name, cookie.value) == ('x', 'y')
True
>>> cookie2 = Cookie.from_string('yak=mov')
>>> (cookie2.name, cookie2.value) == ('yak', 'mov')
True
```

But here there is an important caveat. Since Cookie.from_string only ever returns a Cookie instance, you can't use it to parse request headers which may contain multiple name/value pairs (as in 'Cookie: a=b; c=d'). It would be lame

if your program crashed or did something dumb depending on this sort of difference, so use Cookies.from_request, to ensure you get consistent behavior regardless of your input.

```
>>> try:
...   cookie3 = Cookie.from_string('Cookie: duh=frob')
... except Exception as e:
...   print(type(e))
<class 'cookies.InvalidCookieError'>
>>> from cookies import Cookies
>>> cookies = Cookies.from_request('Cookie: duh=frob')
>>> cookies['duh'].value == 'frob'
True
```

(See the next section for more on Cookies collection objects.)

You can also make a cookie object from a dict that maps attribute names to values. This will parse the values as strings, which can be convenient when you don't have an existing string to parse.

```
>>> from datetime import datetime
>>> cookie = Cookie.from_dict({'name': 'x', 'value': 'y', 'expires': 'Thu, 23 Jan 2003 00:00:00 GMT'
>>> (cookie.name, cookie.value, cookie.expires) == ('x', 'y', datetime(2003, 1, 23, 0, 0))
True
>>> cookie = Cookie.from_dict(dict(name='x', value='y'))
>>> (cookie.name, cookie.value) == ('x', 'y')
True
```

You can also do the reverse operation with to_dict():

```
>>> cookie = Cookie('x', 'y', comment='no')
>>> sorted(cookie.to_dict().items())
[('Comment', 'no'), ('name', 'x'), ('value', 'y')]
```

If you just want the attributes other than name and value, you can export those to a dict with the attributes() method, which produces a mapping of attribute names to encoded values and is also used internally for rendering:

```
>>> cookie.attributes()
{'Comment': 'no'}
```

When you set cookie attributes, the library tries to make sure that it is decoded appropriately during parse, has a usable kind of value in Python, and is encoded appropriately during render. For example, the expires attribute represents a date which might come in in any of many standard and non-standard formats. From Python, it should be a datetime object. When rendering, it should always be produced in the standard format.

The following example uses the parse_date function to create the datetime (though that parsing can also be done indirectly by using from_dict or from_string).

```
>>> from cookies import parse_date
>>> a = Cookie('a', 'blah')
>>> a.expires = parse_date("Wed, 23-Jan-1992 00:01:02 GMT")
>>> a.render_response()
'a=blah; Expires=Thu, 23 Jan 1992 00:01:02 GMT'
```

```
>>> # asctime format is also handled...
>>> b = Cookie('b', 'blr')
>>> b.expires = parse_date("Sun Nov  6 08:49:37 1994")
>>> b.render_response()
'b=blr; Expires=Sun, 06 Nov 1994 08:49:37 GMT'
```

Cookie objects can be meaningfully compared; they are equal or unequal based on their attributes. If one has an attribute the other is missing, they are not equal.

```
>>> x = Cookie('a', 'b')
>>> y = Cookie('a', 'b')
>>> x == y
True
>>> x is y
False
>>> z = Cookie('a', 'b', secure=True)
>>> z.secure == True
True
>>> not x.secure
True
>>> x == z
False
>>> x.name == z.name and x.value == z.value
True
```

Cookies objects

Often you just want to parse a batch of cookies and start looking at them.

The following example shows a typical case: how a web app might handle the value it gets in the HTTP_COOKIE CGI (or WSGI) variable. Since this is a request header, use the from_request() method.

```
>>> from cookies import Cookies
>>> cookies = Cookies.from_request("a=b; c=d; e=f")
```

The resulting Cookies object can be used just like a dict of Cookie objects.

```
>>> sorted(cookies.keys())
['a', 'c', 'e']
>>> 'a' in cookies
True
>>> try:
...   cookies['x']
... except KeyError:
...   print("didn't exist")
didn't exist
>>> a = cookies['a']
>>> # Each item in a Cookies object is a Cookie.
>>> type(a)
<class 'cookies.Cookie'>
>>> del cookies['a']
>>> try: cookies['a']
... except KeyError: print("deleted")
deleted
```

Calling cookies.parse_request() will add more cookies to the same object, so you can build it up incrementally. However, it won't overwrite existing cookies with the same name, to ensure that only the first one is taken.

```
>>> cookies['c'].value == 'd'
True
>>> _ = cookies.parse_request('x=y; c=mumbles')
>>> cookies['x'].value == 'y'
True
>>> cookies['c'].value == 'd'
True
```

You can also use parse_response to add cookies from 'Set-Cookie' response headers in the same incremental way, with the same provisos. (This has to be a different method, because response headers are different from request headers and must be parsed differently.)

```
>>> cookies = Cookies.from_response("Set-Cookie: z=b")
>>> _ = cookies.parse_response("Set-Cookie: y=a")
>>> cookies['z'].value == 'b'
True
>>> cookies['y'].value == 'a'
True
```

If you have some cookie objects that were already produced and should just be added to a dict, or you just want to make some new ones quickly, either or both can be done quickly with the add() method. Ordered arguments to the add() method are interpreted as cookie objects, and added under their names. Keyword arguments are interpreted as values for new cookies to be constructed with the given name.

```
>>> cookies = Cookies()
>>> cookies.add(Cookie('a','b'))
>>> cookies.add(x='y')
>>> cookies.add(Cookie('c','d'), e='f')
>>> sorted(cookies.keys())
['a', 'c', 'e', 'x']
>>> sorted(cookie.value for cookie in cookies.values())
['b', 'd', 'f', 'y']
```

Other than parsing strings into Cookie objects, or modifying them, you might also want to generate rendered output. For this, use render_request() or render_response(), depending on the sort of headers you want to render. You can render all the headers at once - either as separate lines, or all on one line.

```
>>> cookies = Cookies()
>>> cookies.add(Cookie('mom', 'strong'))
>>> cookies.add(Cookie('dad', 'pretty'))
>>> cookies.render_request()
'dad=pretty; mom=strong'
```

Each individual cookie can be rendered either in the format for an HTTP request, or the format for an HTTP response. Attribute values can be manipulated in natural ways and the rendered output changes appropriately; but rendered request headers don't include attributes (as they shouldn't):

```
>>> from datetime import datetime
>>> cookies = Cookies(a='foo', b='bar')
>>> cookies['a'].render_request()
'a=foo'
>>> cookies['b'].max_age = 42
>>> cookies['b'].render_response()
'b=bar; Max-Age=42'
>>> cookies['b'].max_age += 10
>>> cookies['b'].render_response()
'b=bar; Max-Age=52'

# Set attributes on individual cookies.
>>> cookies['a'].expires = datetime(2003, 1, 23, 0, 0, 0)
>>> cookies.add(c='d')
>>> cookies['c'].path = "/"
>>> cookies['c'].path
'/'

# Render request headers
>>> cookies.render_request()
```

```
'a=foo; b=bar; c=d'

# Render response headers - more detail.
>>> rendered = cookies.render_response()
>>> rendered[0]
'a=foo; Expires=Thu, 23 Jan 2003 00:00:00 GMT'
>>> rendered[1]
'b=bar; Max-Age=52'
>>> rendered[2]
'c=d; Path=/'
```

Cookies objects can also be compared to each other: this is the same as comparing all their individual cookies.

```
>>> c1 = Cookies(a='b', c='d')
>>> c2 = Cookies(a='b', c='d')
>>> c3 = Cookies(a='b')
>>> c1 == c2
True
>>> c2 == c3
False
```

Extension Mechanisms

Many aspects of the Cookie class can be customized to get different behavior. For example, new attributes can be supported or existing attributes can be treated differently by changing the attribute_renderers, attribute_parsers, and attribute_validators dicts. See the source for defaults and details.

In addition to the provided extension mechanisms, much of the functionality is exposed in a lower-level imperative API which you can use to do things imperatively or make your own object interfaces. Also, the regexps used in the parser are exposed individually to help you with unusual tasks like writing special tests or handling new attributes. Check out the source for more information.