zope.testbrowser Documentation

Release 5.0

Zope Foundation and Contributors

Jun 20, 2017

Contents

Contents:

CHAPTER 1

Using `zope.testbrowser`

Different Browsers

HTTP Browser

The `zope.testbrowser.browser` module exposes a `Browser` class that simulates a web browser similar to Mozilla Firefox or IE.

```
>>> from zope.testbrowser.browser import Browser
>>> browser = Browser()
```

This version of the browser object can be used to access any web site just as you would do using a normal web browser.

WSGI Test Browser

General usage

There is also a special version of the `Browser` class which uses WebTest and can be used to do functional testing of WSGI applications. It can be imported from `zope.testbrowser.wsgi`:

```
>>> from zope.testbrowser.wsgi import Browser
>>> from zope.testbrowser.testing import demo_app
>>> browser = Browser('http://localhost/', wsgi_app=demo_app)
>>> print(browser.contents)
Hello world!
...
```

You can also use it with zope layers if you

- write a subclass of `zope.testbrowser.wsgi.Layer` and override the `make_wsgi_app` method, then
- use an instance of the class as the test layer of your test.

Example:

```
>>> import zope.testbrowser.wsgi
>>> class SimpleLayer(zope.testbrowser.wsgi.Layer):
...     def make_wsgi_app(self):
...         return simple_app
```

Where `simple_app` is the callable of your WSGI application.

Testing a Zope 2/Zope 3/Bluebream WSGI application

When testing a Zope 2/Zope 3/Bluebream WSGI application you should wrap your WSGI application under test into `zope.testbrowser.wsgi.AuthorizationMiddleware` as all these application servers expect basic authentication headers to be base64 encoded. This middleware handles this for you.

Example when using the layer:

```
>>> import zope.testbrowser.wsgi
>>> class ZopeSimpleLayer(zope.testbrowser.wsgi.Layer):
...     def make_wsgi_app(self):
...         return zope.testbrowser.wsgi.AuthorizationMiddleware(simple_app)
```

There is also a `BrowserLayer` in zope.app.wsgi.testlayer which does this for you and includes a `TransactionMiddleware`, too, which could be handy when testing a ZODB based application.

However, since the `BrowserLayer` in zope.app.wsgi.testlayer re-creates the ZODB in `testSetUp`, we need to re-create the WSGI App during `testSetUp`, too. Therefore use `TestBrowserLayer` of `zope.testbrowser.wsgi` instead of the simpler `Layer` to combine it with the `BrowserLayer` in zope.app.wsgi.testlayer:

```
>>> import zope.testbrowser.wsgi
>>> import zope.app.wsgi.testlayer
>>> class Layer(zope.testbrowser.wsgi.TestBrowserLayer,
...             zope.app.wsgi.testlayer.BrowserLayer):
...     pass
```

Bowser Usage

We will test this browser against a WSGI test application:

```
>>> from zope.testbrowser.ftests.wsgitestapp import WSGITestApplication
>>> wsgi_app = WSGITestApplication()
```

An initial page to load can be passed to the `Browser` constructor:

```
>>> browser = Browser('http://localhost/@@/testbrowser/simple.html', wsgi_app=wsgi_
↪app)
>>> browser.url
'http://localhost/@@/testbrowser/simple.html'
```

The browser can send arbitrary headers; this is helpful for setting the "Authorization" header or a language value, so that your tests format values the way you expect in your tests, if you rely on zope.i18n locale-based formatting or a similar approach.

```
>>> browser.addHeader('Authorization', 'Basic mgr:mgrpw')
>>> browser.addHeader('Accept-Language', 'en-US')
```

An existing browser instance can also *open* web pages:

```
>>> browser.open('http://localhost/@@/testbrowser/simple.html')
>>> browser.url
'http://localhost/@@/testbrowser/simple.html'
```

Once you have opened a web page initially, best practice for writing testbrowser doctests suggests using 'click' to navigate further (as discussed below), except in unusual circumstances.

The test browser complies with the IBrowser interface; see `zope.testbrowser.interfaces` for full details on the interface.

```
>>> from zope.testbrowser import interfaces
>>> from zope.interface.verify import verifyObject
>>> verifyObject(interfaces.IBrowser, browser)
True
```

Page Contents

The contents of the current page are available:

```
 >>> print(browser.contents)
 <html>
   <head>
     <title>Simple Page</title>
   </head>
   <body>
     <h1>Simple Page</h1>
   </body>
 </html>
```

Making assertions about page contents is easy.

```
>>> '<h1>Simple Page</h1>' in browser.contents
True
```

Utilizing the doctest facilities, it also possible to do:

```
>>> browser.contents
'...<h1>Simple Page</h1>...'
```

Note: Unfortunately, ellipsis (...) cannot be used at the beginning of the output (this is a limitation of doctest).

Checking for HTML

Not all URLs return HTML. Of course our simple page does:

```
>>> browser.open('http://localhost/@@/testbrowser/simple.html')
>>> browser.isHtml
True
```

But if we load an image (or other binary file), we do not get HTML:

```
>>> browser.open('http://localhost/@@/testbrowser/zope3logo.gif')
>>> browser.isHtml
False
```

HTML Page Title

Another useful helper property is the title:

```
>>> browser.open('http://localhost/@@/testbrowser/simple.html')
>>> browser.title
'Simple Page'
```

If a page does not provide a title, it is simply `None`:

```
>>> browser.open('http://localhost/@@/testbrowser/notitle.html')
>>> browser.title
```

However, if the output is not HTML, then an error will occur trying to access the title:

```
>>> browser.open('http://localhost/@@/testbrowser/zope3logo.gif')
>>> browser.title
Traceback (most recent call last):
...
BrowserStateError: not viewing HTML
```

Headers

As you can see, the *contents* of the browser does not return any HTTP headers. The headers are accessible via a separate attribute, which is an `httplib.HTTPMessage` instance (httplib is a part of Python's standard library):

```
>>> browser.open('http://localhost/@@/testbrowser/simple.html')
>>> browser.headers
<httplib.HTTPMessage instance...>
```

The headers can be accessed as a string:

```
>>> print(browser.headers)
Status: 200 OK
Content-Length: 109
Content-Type: text/html; charset=UTF-8
```

Or as a mapping:

```
>>> browser.headers['content-type']
'text/html; charset=UTF-8'
```

Cookies

When a Set-Cookie header is available, it can be found in the headers, as seen above. Here, we use a view that will make the server set cookies with the values we provide.

```
>>> browser.open('http://localhost/set_cookie.html?name=foo&value=bar')
>>> browser.headers['set-cookie'].replace(';', '')
'foo=bar'
```

It is also available in the browser's `cookies` attribute. This is an extended mapping interface that allows getting, setting, and deleting the cookies that the browser is remembering *for the current url*. Here are a few examples.

```
>>> browser.cookies['foo']
'bar'
>>> browser.cookies.keys()
['foo']
>>> list(browser.cookies.values())
['bar']
>>> list(browser.cookies.items())
[('foo', 'bar')]
>>> 'foo' in browser.cookies
True
>>> 'bar' in browser.cookies
False
>>> len(browser.cookies)
1
>>> print(dict(browser.cookies))
{'foo': 'bar'}
>>> browser.cookies['sha'] = 'zam'
>>> len(browser.cookies)
2
>>> sorted(browser.cookies.items())
[('foo', 'bar'), ('sha', 'zam')]
>>> browser.open('http://localhost/get_cookie.html')
>>> print(browser.headers.get('set-cookie'))
None
>>> print(browser.contents) # server got the cookie change
foo: bar
sha: zam
>>> sorted(browser.cookies.items())
[('foo', 'bar'), ('sha', 'zam')]
>>> browser.cookies.clearAll()
>>> len(browser.cookies)
0
```

Many more examples, and a discussion of the additional methods available, can be found in cookies.txt.

Navigation and Link Objects

If you want to simulate clicking on a link, get the link and *click* on it. In the *navigate.html* file there are several links set up to demonstrate the capabilities of the link objects and their *click* method.

The simplest way to get a link is via the anchor text. In other words the text you would see in a browser (text and url searches are substring searches):

```
>>> browser.open('http://localhost/@@/testbrowser/navigate.html')
>>> browser.contents
'...<a href="navigate.html?message=By+Link+Text">Link Text</a>...'
>>> link = browser.getLink('Link Text')
>>> link
<Link text='Link Text' url='http://localhost/@@/testbrowser/navigate.html?
↪message=By+Link+Text'>
```

Link objects comply with the ILink interface.

```
>>> verifyObject(interfaces.ILink, link)
True
```

Links expose several attributes for easy access.

```
>>> link.text
'Link Text'
>>> link.tag # links can also be image maps.
'a'
>>> link.url # it's normalized
'http://localhost/@@/testbrowser/navigate.html?message=By+Link+Text'
>>> link.attrs
{'href': 'navigate.html?message=By+Link+Text'}
```

Links can be "clicked" and the browser will navigate to the referenced URL.

```
>>> link.click()
>>> browser.url
'http://localhost/@@/testbrowser/navigate.html?message=By+Link+Text'
>>> browser.contents
'...Message: <em>By Link Text</em>...'
```

When finding a link by its text, whitespace is normalized.

```
>>> browser.open('http://localhost/@@/testbrowser/navigate.html')
>>> browser.contents
'...> Link Text \n    with     Whitespace\tNormalization (and parens) </...'
>>> link = browser.getLink('Link Text with Whitespace Normalization '
...                        '(and parens)')
>>> link
<Link text='Link Text with Whitespace Normalization (and parens)'...>
>>> link.text
'Link Text with Whitespace Normalization (and parens)'
>>> link.click()
>>> browser.url
'http://localhost/@@/testbrowser/navigate.html?message=By+Link+Text+with+Normalization
↪'
>>> browser.contents
'...Message: <em>By Link Text with Normalization</em>...'
```

When a link text matches more than one link, by default the first one is chosen. You can, however, specify the index of the link and thus retrieve a later matching link:

```
>>> browser.getLink('Link Text')
<Link text='Link Text' ...>

>>> browser.getLink('Link Text', index=1)
<Link text='Link Text with Whitespace Normalization (and parens)' ...>
```

Note that clicking a link object after its browser page has expired will generate an error.

```
>>> link.click()
Traceback (most recent call last):
...
ExpiredError
```

You can also find the link by its URL,

```
>>> browser.open('http://localhost/@@/testbrowser/navigate.html')
>>> browser.contents
'...<a href="navigate.html?message=By+URL">Using the URL</a>...'

>>> browser.getLink(url='?message=By+URL').click()
>>> browser.url
'http://localhost/@@/testbrowser/navigate.html?message=By+URL'
>>> browser.contents
'...Message: <em>By URL</em>...'
```

or its id:

```
>>> browser.open('http://localhost/@@/testbrowser/navigate.html')
>>> browser.contents
'...<a href="navigate.html?message=By+Id" id="anchorid">By Anchor Id</a>...'

>>> browser.getLink(id='anchorid').click()
>>> browser.url
'http://localhost/@@/testbrowser/navigate.html?message=By+Id'
>>> browser.contents
'...Message: <em>By Id</em>...'
```

You thought we were done here? Not so quickly. The *getLink* method also supports image maps, though not by specifying the coordinates, but using the area's id:

```
>>> browser.open('http://localhost/@@/testbrowser/navigate.html')
>>> link = browser.getLink(id='zope3')
>>> link.tag
'area'
>>> link.click()
>>> browser.url
'http://localhost/@@/testbrowser/navigate.html?message=Zope+3+Name'
>>> browser.contents
'...Message: <em>Zope 3 Name</em>...'
```

Getting a nonexistent link raises an exception.

```
>>> browser.open('http://localhost/@@/testbrowser/navigate.html')
>>> browser.getLink('This does not exist')
Traceback (most recent call last):
...
LinkNotFoundError
```

A convenience method is provided to follow links; this uses the same arguments as *getLink*, but clicks on the link instead of returning the link object.

```
>>> browser.open('http://localhost/@@/testbrowser/navigate.html')
>>> browser.contents
'...<a href="navigate.html?message=By+Link+Text">Link Text</a>...'
>>> browser.follow('Link Text')
>>> browser.url
'http://localhost/@@/testbrowser/navigate.html?message=By+Link+Text'
>>> browser.contents
'...Message: <em>By Link Text</em>...'

>>> browser.open('http://localhost/@@/testbrowser/navigate.html')
```

```
>>> browser.follow(url='?message=By+URL')
>>> browser.url
'http://localhost/@@/testbrowser/navigate.html?message=By+URL'
>>> browser.contents
'...Message: <em>By URL</em>...'

>>> browser.open('http://localhost/@@/testbrowser/navigate.html')
>>> browser.follow(id='zope3')
>>> browser.url
'http://localhost/@@/testbrowser/navigate.html?message=Zope+3+Name'
>>> browser.contents
'...Message: <em>Zope 3 Name</em>...'
```

Attempting to follow links that don't exist raises the same exception as asking for the link object:

```
>>> browser.follow('This does not exist')
Traceback (most recent call last):
...
LinkNotFoundError
```

Other Navigation

Like in any normal browser, you can reload a page:

```
>>> browser.open('http://localhost/@@/testbrowser/simple.html')
>>> browser.url
'http://localhost/@@/testbrowser/simple.html'
>>> browser.reload()
>>> browser.url
'http://localhost/@@/testbrowser/simple.html'
```

You can also go back:

```
>>> browser.open('http://localhost/@@/testbrowser/notitle.html')
>>> browser.url
'http://localhost/@@/testbrowser/notitle.html'
>>> browser.goBack()
>>> browser.url
'http://localhost/@@/testbrowser/simple.html'
```

Controls

One of the most important features of the browser is the ability to inspect and fill in values for the controls of input forms. To do so, let's first open a page that has a bunch of controls:

```
>>> browser.open('http://localhost/@@/testbrowser/controls.html')
```

Obtaining a Control

You look up browser controls with the 'getControl' method. The default first argument is 'label', and looks up the form on the basis of any associated label.

```
>>> control = browser.getControl('Text Control')
>>> control
<Control name='text-value' type='text'>
>>> browser.getControl(label='Text Control') # equivalent
<Control name='text-value' type='text'>
```

If you request a control that doesn't exist, the code raises a LookupError:

```
>>> browser.getControl('Does Not Exist')
Traceback (most recent call last):
...
LookupError: label 'Does Not Exist'
available items:
  <TextControl(text-value=Some Text)>
  <PasswordControl(password-value=Password)>
  <HiddenControl(hidden-value=Hidden) (readonly)>
  ...
```

If you request a control with an ambiguous lookup, the code raises an AmbiguityError.

```
>>> browser.getControl('Ambiguous Control')
Traceback (most recent call last):
...
AmbiguityError: label 'Ambiguous Control' matches:
  <TextControl(ambiguous-control-name=First)>
  <TextControl(ambiguous-control-name=Second)>
```

This is also true if an option in a control is ambiguous in relation to the control itself.

```
>>> browser.getControl('Sub-control Ambiguity')
Traceback (most recent call last):
...
AmbiguityError: label 'Sub-control Ambiguity' matches:
  <SelectControl(ambiguous-subcontrol=[*, ambiguous])>
  <Item name='ambiguous' id=None contents='Sub-control Ambiguity Exemplified' value=
↪'ambiguous' label='Sub-control Ambiguity Exemplified'>
```

Ambiguous controls may be specified using an index value. We use the control's value attribute to show the two controls; this attribute is properly introduced below.

```
>>> browser.getControl('Ambiguous Control', index=0)
<Control name='ambiguous-control-name' type='text'>
>>> browser.getControl('Ambiguous Control', index=0).value
'First'
>>> browser.getControl('Ambiguous Control', index=1).value
'Second'
>>> browser.getControl('Sub-control Ambiguity', index=0)
<ListControl name='ambiguous-subcontrol' type='select'>
>>> browser.getControl('Sub-control Ambiguity', index=1).optionValue
'ambiguous'
>>> browser.getControl('Sub-control Ambiguity', index=2)
Traceback (most recent call last):
...
LookupError: label 'Sub-control Ambiguity'
Index 2 out of range, available choices are 0...1
  0: <SelectControl(ambiguous-subcontrol=[*, ambiguous])>
  1: <Item name='ambiguous' id=None contents='Sub-control Ambiguity Exemplified'␣
↪value='ambiguous' label='Sub-control Ambiguity Exemplified'>
```

Label searches are against stripped, whitespace-normalized, no-tag versions of the text. Text applied to searches is also stripped and whitespace normalized. The search finds results if the text search finds the whole words of your text in a label. Thus, for instance, a search for 'Add' will match the label 'Add a Client' but not 'Address'. Case is honored.

```
>>> browser.getControl('Label Needs Whitespace Normalization')
<Control name='label-needs-normalization' type='text'>
>>> browser.getControl('label needs whitespace normalization')
Traceback (most recent call last):
...
LookupError: label 'label needs whitespace normalization'
...
>>> browser.getControl(' Label  Needs Whitespace    ')
<Control name='label-needs-normalization' type='text'>
>>> browser.getControl('Whitespace')
<Control name='label-needs-normalization' type='text'>
>>> browser.getControl('hitespace')
Traceback (most recent call last):
...
LookupError: label 'hitespace'
...
>>> browser.getControl('[non word characters should not confuse]')
<Control name='non-word-characters' type='text'>
```

Multiple labels can refer to the same control (simply because that is possible in the HTML 4.0 spec).

```
>>> browser.getControl('Multiple labels really')
<Control name='two-labels' type='text'>
>>> browser.getControl('really are possible')
<Control name='two-labels' type='text'>
>>> browser.getControl('really') # OK: ambiguous labels, but not ambiguous control
<Control name='two-labels' type='text'>
```

A label can be connected with a control using the 'for' attribute and also by containing a control.

```
>>> browser.getControl(
...     'Labels can be connected by containing their respective fields')
<Control name='contained-in-label' type='text'>
```

Get also accepts one other search argument, 'name'. Only one of 'label' and 'name' may be used at a time. The 'name' keyword searches form field names.

```
>>> browser.getControl(name='text-value')
<Control name='text-value' type='text'>
>>> browser.getControl(name='ambiguous-control-name')
Traceback (most recent call last):
...
AmbiguityError: name 'ambiguous-control-name' matches:
  <TextControl(ambiguous-control-name=First)>
  <TextControl(ambiguous-control-name=Second)>
>>> browser.getControl(name='does-not-exist')
Traceback (most recent call last):
...
LookupError: name 'does-not-exist'
available items:
  <TextControl(text-value=Some Text)>
  ...
>>> browser.getControl(name='ambiguous-control-name', index=1).value
'Second'
```

Combining 'label' and 'name' raises a ValueError, as does supplying neither of them.

```
>>> browser.getControl(label='Ambiguous Control', name='ambiguous-control-name')
Traceback (most recent call last):
...
ValueError: Supply one and only one of "label" and "name" as arguments
>>> browser.getControl()
Traceback (most recent call last):
...
ValueError: Supply one and only one of "label" and "name" as arguments
```

Radio and checkbox fields are unusual in that their labels and names may point to different objects: names point to logical collections of radio buttons or checkboxes, but labels may only be used for individual choices within the logical collection. This means that obtaining a radio button by label gets a different object than obtaining the radio collection by name. Select options may also be searched by label.

```
>>> browser.getControl(name='radio-value')
<ListControl name='radio-value' type='radio'>
>>> browser.getControl('Zwei')
<ItemControl name='radio-value' type='radio' optionValue='2' selected=True>
>>> browser.getControl('One')
<ItemControl name='multi-checkbox-value' type='checkbox' optionValue='1'␣
↪selected=True>
>>> browser.getControl('Tres')
<ItemControl name='single-select-value' type='select' optionValue='3' selected=False>
```

Characteristics of controls and subcontrols are discussed below.

Control Objects

Controls provide IControl.

```
>>> ctrl = browser.getControl('Text Control')
>>> ctrl
<Control name='text-value' type='text'>
>>> verifyObject(interfaces.IControl, ctrl)
True
```

They have several useful attributes:

- the name as which the control is known to the form:

```
>>> ctrl.name
'text-value'
```

- the value of the control, which may also be set:

```
>>> ctrl.value
'Some Text'
>>> ctrl.value = 'More Text'
>>> ctrl.value
'More Text'
```

- the type of the control:

```
>>> ctrl.type
'text'
```

- a flag describing whether the control is disabled:

```
>>> ctrl.disabled
False
```

- and a flag to tell us whether the control can have multiple values:

```
>>> ctrl.multiple
False
```

Additionally, controllers for select, radio, and checkbox provide IListControl. These fields have four other attributes and an additional method:

```
>>> ctrl = browser.getControl('Multiple Select Control')
>>> ctrl
<ListControl name='multi-select-value' type='select'>
>>> ctrl.disabled
False
>>> ctrl.multiple
True
>>> verifyObject(interfaces.IListControl, ctrl)
True
```

- 'options' lists all available value options.

```
>>> ctrl.options
['1', '2', '3']
```

- 'displayOptions' lists all available options by label. The 'label' attribute on an option has precedence over its contents, which is why our last option is 'Third' in the display.

```
>>> ctrl.displayOptions
['Un', 'Deux', 'Third']
```

- 'displayValue' lets you get and set the displayed values of the control of the select box, rather than the actual values.

```
>>> ctrl.value
[]
>>> ctrl.displayValue
[]
>>> ctrl.displayValue = ['Un', 'Deux']
>>> ctrl.displayValue
['Un', 'Deux']
>>> ctrl.value
['1', '2']
```

- 'controls' gives you a list of the subcontrol objects in the control (subcontrols are discussed below).

```
 >>> ctrl.controls
 [<ItemControl name='multi-select-value' type='select' optionValue='1' selected=True>,
  <ItemControl name='multi-select-value' type='select' optionValue='2' selected=True>,
  <ItemControl name='multi-select-value' type='select' optionValue='3' selected=False>
↪]
```

- The 'getControl' method lets you get subcontrols by their label or their value.

```
>>> ctrl.getControl('Un')
<ItemControl name='multi-select-value' type='select' optionValue='1' selected=True>
>>> ctrl.getControl('Deux')
<ItemControl name='multi-select-value' type='select' optionValue='2' selected=True>
>>> ctrl.getControl('Trois') # label attribute
<ItemControl name='multi-select-value' type='select' optionValue='3' selected=False>
>>> ctrl.getControl('Third') # contents
<ItemControl name='multi-select-value' type='select' optionValue='3' selected=False>
>>> browser.getControl('Third') # ambiguous in the browser, so useful
Traceback (most recent call last):
...
AmbiguityError: label 'Third' matches:
  <Item name='3' id=None contents='Tres' value='3' label='Third'>
  <Item name='3' id=None contents='Trois' value='3' label='Third'>
  <Item name='3' id='multi-checkbox-value-3' __label={'__text': 'Three\n        '}␣
↪checked='checked' name='multi-checkbox-value' type='checkbox' id='multi-checkbox-
↪value-3' value='3'>
  <Item name='3' id='radio-value-3' __label={'__text': ' Drei'} type='radio' name=
↪'radio-value' value='3' id='radio-value-3'>
```

Finally, submit controls provide `ISubmitControl`, and image controls provide `IImageSubmitControl`, which extents `ISubmitControl`. These both simply add a 'click' method. For image submit controls, you may also provide a coordinates argument, which is a tuple of (x, y). These submit the forms, and are demonstrated below as we examine each control individually.

ItemControl Objects

As introduced briefly above, using labels to obtain elements of a logical radio button or checkbox collection returns item controls, which are parents. Manipulating the value of these controls affects the parent control.

```
>>> browser.getControl(name='radio-value').value
['2']
>>> browser.getControl('Zwei').optionValue # read-only.
'2'
>>> browser.getControl('Zwei').selected
True
>>> verifyObject(interfaces.IItemControl, browser.getControl('Zwei'))
True
>>> browser.getControl('Ein').selected = True
>>> browser.getControl('Ein').selected
True
>>> browser.getControl('Zwei').selected
False
>>> browser.getControl(name='radio-value').value
['1']
>>> browser.getControl('Ein').selected = False
>>> browser.getControl(name='radio-value').value
[]
>>> browser.getControl('Zwei').selected = True
```

Checkbox collections behave similarly, as shown below.

Various Controls

The various types of controls are demonstrated here.

Text Control

The text control we already introduced above.

Password Control

```
>>> ctrl = browser.getControl('Password Control')
>>> ctrl
<Control name='password-value' type='password'>
>>> verifyObject(interfaces.IControl, ctrl)
True
>>> ctrl.value
'Password'
>>> ctrl.value = 'pass now'
>>> ctrl.value
'pass now'
>>> ctrl.disabled
False
>>> ctrl.multiple
False
```

Hidden Control

```
>>> ctrl = browser.getControl(name='hidden-value')
>>> ctrl
<Control name='hidden-value' type='hidden'>
>>> verifyObject(interfaces.IControl, ctrl)
True
>>> ctrl.value
'Hidden'
>>> ctrl.value = 'More Hidden'
>>> ctrl.disabled
False
>>> ctrl.multiple
False
```

Text Area Control

```
>>> ctrl = browser.getControl('Text Area Control')
>>> ctrl
<Control name='textarea-value' type='textarea'>
>>> verifyObject(interfaces.IControl, ctrl)
True
>>> ctrl.value
'        Text inside\n        area!\n      '
>>> ctrl.value = 'A lot of\n text.'
>>> ctrl.disabled
False
>>> ctrl.multiple
False
```

File Control

File controls are used when a form has a file-upload field. To specify data, call the add_file method, passing:

- A file-like object
- a content type, and
- a file name

```
>>> ctrl = browser.getControl('File Control')
>>> ctrl
<Control name='file-value' type='file'>
>>> verifyObject(interfaces.IControl, ctrl)
True
>>> ctrl.value is None
True
>>> import io

>>> ctrl.add_file(io.BytesIO(b'File contents'),
...               'text/plain', 'test.txt')

The file control (like the other controls) also knows if it is disabled
or if it can have multiple values.

>>> ctrl.disabled
False
>>> ctrl.multiple
False
```

Selection Control (Single-Valued)

```
>>> ctrl = browser.getControl('Single Select Control')
>>> ctrl
<ListControl name='single-select-value' type='select'>
>>> verifyObject(interfaces.IListControl, ctrl)
True
>>> ctrl.value
['1']
>>> ctrl.value = ['2']
>>> ctrl.disabled
False
>>> ctrl.multiple
False
>>> ctrl.options
['1', '2', '3']
>>> ctrl.displayOptions
['Uno', 'Dos', 'Third']
>>> ctrl.displayValue
['Dos']
>>> ctrl.displayValue = ['Tres']
>>> ctrl.displayValue
['Third']
>>> ctrl.displayValue = ['Dos']
>>> ctrl.displayValue
['Dos']
>>> ctrl.displayValue = ['Third']
```

```
>>> ctrl.displayValue
['Third']
>>> ctrl.value
['3']
```

Selection Control (Multi-Valued)

This was already demonstrated in the introduction to control objects above.

Checkbox Control (Single-Valued; Unvalued)

```
>>> ctrl = browser.getControl(name='single-unvalued-checkbox-value')
>>> ctrl
<ListControl name='single-unvalued-checkbox-value' type='checkbox'>
>>> verifyObject(interfaces.IListControl, ctrl)
True
>>> ctrl.value
True
>>> ctrl.value = False
>>> ctrl.disabled
False
>>> ctrl.multiple
True
>>> ctrl.options
[True]
>>> ctrl.displayOptions
['Single Unvalued Checkbox']
>>> ctrl.displayValue
[]
>>> verifyObject(
...     interfaces.IItemControl,
...     browser.getControl('Single Unvalued Checkbox'))
True
>>> browser.getControl('Single Unvalued Checkbox').optionValue
'on'
>>> browser.getControl('Single Unvalued Checkbox').selected
False
>>> ctrl.displayValue = ['Single Unvalued Checkbox']
>>> ctrl.displayValue
['Single Unvalued Checkbox']
>>> browser.getControl('Single Unvalued Checkbox').selected
True
>>> browser.getControl('Single Unvalued Checkbox').selected = False
>>> browser.getControl('Single Unvalued Checkbox').selected
False
>>> ctrl.displayValue
[]
>>> browser.getControl(
...     name='single-disabled-unvalued-checkbox-value').disabled
True
```

Checkbox Control (Single-Valued, Valued)

```
  >>> ctrl = browser.getControl(name='single-valued-checkbox-value')
  >>> ctrl
  <ListControl name='single-valued-checkbox-value' type='checkbox'>
  >>> verifyObject(interfaces.IListControl, ctrl)
  True
  >>> ctrl.value
  ['1']
  >>> ctrl.value = []
  >>> ctrl.disabled
  False
  >>> ctrl.multiple
  True
  >>> ctrl.options
  ['1']
  >>> ctrl.displayOptions
  ['Single Valued Checkbox']
  >>> ctrl.displayValue
  []
  >>> verifyObject(
  ...     interfaces.IItemControl,
  ...     browser.getControl('Single Valued Checkbox'))
  True
  >>> browser.getControl('Single Valued Checkbox').selected
  False
  >>> browser.getControl('Single Valued Checkbox').optionValue
  '1'
  >>> ctrl.displayValue = ['Single Valued Checkbox']
  >>> ctrl.displayValue
  ['Single Valued Checkbox']
  >>> browser.getControl('Single Valued Checkbox').selected
  True
  >>> browser.getControl('Single Valued Checkbox').selected = False
  >>> browser.getControl('Single Valued Checkbox').selected
  False
  >>> ctrl.displayValue
  []

- Checkbox Control (Multi-Valued)

  >>> ctrl = browser.getControl(name='multi-checkbox-value')
  >>> ctrl
  <ListControl name='multi-checkbox-value' type='checkbox'>
  >>> verifyObject(interfaces.IListControl, ctrl)
  True
  >>> ctrl.value
  ['1', '3']
  >>> ctrl.value = ['1', '2']
  >>> ctrl.disabled
  False
  >>> ctrl.multiple
  True
  >>> ctrl.options
  ['1', '2', '3']
  >>> ctrl.displayOptions
  ['One', 'Two', 'Three']
  >>> ctrl.displayValue
```

```
['One', 'Two']
>>> ctrl.displayValue = ['Two']
>>> ctrl.value
['2']
>>> browser.getControl('Two').optionValue
'2'
>>> browser.getControl('Two').selected
True
>>> verifyObject(interfaces.IItemControl, browser.getControl('Two'))
True
>>> browser.getControl('Three').selected = True
>>> browser.getControl('Three').selected
True
>>> browser.getControl('Two').selected
True
>>> ctrl.value
['2', '3']
>>> browser.getControl('Two').selected = False
>>> ctrl.value
['3']
>>> browser.getControl('Three').selected = False
>>> ctrl.value
[]
```

Radio Control

This is how you get a radio button based control:

```
>>> ctrl = browser.getControl(name='radio-value')
```

This shows the existing value of the control, as it was in the HTML received from the server:

```
>>> ctrl.value
['2']
```

We can then unselect it:

```
>>> ctrl.value = []
>>> ctrl.value
[]
```

We can also reselect it:

```
>>> ctrl.value = ['2']
>>> ctrl.value
['2']
```

displayValue shows the text the user would see next to the control:

```
>>> ctrl.displayValue
['Zwei']
```

This is just unit testing:

```
>>> ctrl
<ListControl name='radio-value' type='radio'>
```

```
>>> verifyObject(interfaces.IListControl, ctrl)
True
>>> ctrl.disabled
False
>>> ctrl.multiple
False
>>> ctrl.options
['1', '2', '3']
>>> ctrl.displayOptions
['Ein', 'Zwei', 'Drei']
>>> ctrl.displayValue = ['Ein']
>>> ctrl.value
['1']
>>> ctrl.displayValue
['Ein']
```

The radio control subcontrols were illustrated above.

Image Control

```
>>> ctrl = browser.getControl(name='image-value')
>>> ctrl
<ImageControl name='image-value' type='image'>
>>> verifyObject(interfaces.IImageSubmitControl, ctrl)
True
>>> ctrl.value
''
>>> ctrl.disabled
False
>>> ctrl.multiple
False
```

Submit Control

```
>>> ctrl = browser.getControl(name='submit-value')
>>> ctrl
<SubmitControl name='submit-value' type='submit'>
>>> browser.getControl('Submit This') # value of submit button is a label
<SubmitControl name='submit-value' type='submit'>
>>> browser.getControl('Standard Submit Control') # label tag is legal
<SubmitControl name='submit-value' type='submit'>
>>> browser.getControl('Submit') # multiple labels, but same control
<SubmitControl name='submit-value' type='submit'>
>>> verifyObject(interfaces.ISubmitControl, ctrl)
True
>>> ctrl.value
'Submit This'
>>> ctrl.disabled
False
>>> ctrl.multiple
False
```

Using Submitting Controls

Both the submit and image type should be clickable and submit the form:

```
>>> browser.getControl('Text Control').value = 'Other Text'
>>> browser.getControl('Submit').click()
>>> print(browser.contents)
<html>
...
<em>Other Text</em>
<input type="text" name="text-value" id="text-value" value="Some Text" />
...
<em>Submit This</em>
<input type="submit" name="submit-value" id="submit-value" value="Submit This" />
...
</html>
```

Note that if you click a submit object after the associated page has expired, you will get an error.

```
>>> browser.open('http://localhost/@@/testbrowser/controls.html')
>>> ctrl = browser.getControl('Submit')
>>> ctrl.click()
>>> ctrl.click()
Traceback (most recent call last):
...
ExpiredError
```

All the above also holds true for the image control:

```
>>> browser.open('http://localhost/@@/testbrowser/controls.html')
>>> browser.getControl('Text Control').value = 'Other Text'
>>> browser.getControl(name='image-value').click()
>>> print(browser.contents)
<html>
...
<em>Other Text</em>
<input type="text" name="text-value" id="text-value" value="Some Text" />
...
<em>1</em>
<em>1</em>
<input type="image" name="image-value" id="image-value"
       src="zope3logo.gif" />
...
</html>

>>> browser.open('http://localhost/@@/testbrowser/controls.html')
>>> ctrl = browser.getControl(name='image-value')
>>> ctrl.click()
>>> ctrl.click()
Traceback (most recent call last):
...
ExpiredError
```

But when sending an image, you can also specify the coordinate you clicked:

```
>>> browser.open('http://localhost/@@/testbrowser/controls.html')
>>> browser.getControl(name='image-value').click((50,25))
>>> print(browser.contents)
```

```
<html>
...
<em>50</em>
<em>25</em>
<input type="image" name="image-value" id="image-value"
       src="zope3logo.gif" />
...
</html>
```

Pages Without Controls

What would happen if we tried to look up a control on a page that has none?

```
>>> browser.open('http://localhost/@@/testbrowser/simple.html')
>>> browser.getControl('anything')
Traceback (most recent call last):
...
LookupError: label 'anything'
(there are no form items in the HTML)
```

Forms

Because pages can have multiple forms with like-named controls, it is sometimes necessary to access forms by name or id. The browser's *forms* attribute can be used to do so. The key value is the form's name or id. If more than one form has the same name or id, the first one will be returned.

```
>>> browser.open('http://localhost/@@/testbrowser/forms.html')
>>> form = browser.getForm(name='one')
```

Form instances conform to the IForm interface.

```
>>> verifyObject(interfaces.IForm, form)
True
```

The form exposes several attributes related to forms:

- The name of the form:

```
>>> form.name
'one'
```

- The id of the form:

```
>>> form.id
'1'
```

- The action (target URL) when the form is submitted:

```
>>> form.action
'http://localhost/@@/testbrowser/forms.html'
```

- The method (HTTP verb) used to transmit the form data:

```
>>> form.method
'GET'
```

Besides those attributes, you have also a couple of methods. Like for the browser, you can get control objects, but limited to the current form...

```
>>> form.getControl(name='text-value')
<Control name='text-value' type='text'>
```

...and submit the form.

```
 >>> form.submit('Submit')
 >>> print(browser.contents)
 <html>
 ...
 <em>First Text</em>
 ...
 </html>
```

Submitting also works without specifying a control, as shown below, which is it's primary reason for existing in competition with the control submission discussed above.

Now let me show you briefly that looking up forms is sometimes important. In the *forms.html* template, we have four forms all having a text control named *text-value*. Now, if I use the browser's *get* method,

```
>>> browser.getControl(name='text-value')
Traceback (most recent call last):
...
AmbiguityError: name 'text-value' matches:
  <TextControl(text-value=First Text)>
  <TextControl(text-value=Second Text)>
  <TextControl(text-value=Third Text)>
  <TextControl(text-value=Fourth Text)>
>>> browser.getControl('Text Control')
Traceback (most recent call last):
...
AmbiguityError: label 'Text Control' matches:
  <TextControl(text-value=Third Text)>
  <TextControl(text-value=Fourth Text)>
```

I'll always get an ambiguous form field. I can use the index argument, or with the *getForm* method I can disambiguate by searching only within a given form:

```
>>> form = browser.getForm('2')
>>> form.getControl(name='text-value').value
'Second Text'
>>> form.submit('Submit')
>>> browser.contents
'...<em>Second Text</em>...'
>>> form = browser.getForm('2')
>>> form.getControl('Submit').click()
>>> browser.contents
'...<em>Second Text</em>...'
>>> browser.getForm('3').getControl('Text Control').value
'Third Text'
```

The last form on the page does not have a name, an id, or a submit button. Working with it is still easy, thanks to a index attribute that guarantees order. (Forms without submit buttons are sometimes useful for JavaScript.)

```
>>> form = browser.getForm(index=3)
>>> form.submit()
>>> browser.contents
'...<em>Fourth Text</em>...<em>Submitted without the submit button.</em>...'
```

If a form is requested that does not exists, an exception will be raised.

```
>>> form = browser.getForm('does-not-exist')
Traceback (most recent call last):
LookupError
```

If the HTML page contains only one form, no arguments to *getForm* are needed:

```
>>> oneform = Browser(wsgi_app=wsgi_app)
>>> oneform.open('http://localhost/@@/testbrowser/oneform.html')
>>> form = oneform.getForm()
```

If the HTML page contains more than one form, *index* is needed to disambiguate if no other arguments are provided:

```
>>> browser.getForm()
Traceback (most recent call last):
ValueError: if no other arguments are given, index is required.
```

Submitting a posts body directly

In addition to the open method, Browser has a `post` method that allows a request body to be supplied. This method is particularly helpful when testing AJAX methods.

Let's visit a page that echos some interesting values from it's request:

```
>>> browser.open('http://localhost/echo.html')
>>> print(browser.contents)
HTTP_ACCEPT_LANGUAGE: en-US
HTTP_CONNECTION: close
HTTP_HOST: localhost
HTTP_USER_AGENT: Python-urllib/2.4
PATH_INFO: /echo.html
REQUEST_METHOD: GET
Body: ''
```

Now, we'll try a post. The post method takes a URL, a data string, and an optional content type. If we just pass a string, then a URL-encoded query string is assumed:

```
>>> browser.post('http://localhost/echo.html', 'x=1&y=2')
>>> print(browser.contents)
CONTENT_LENGTH: 7
CONTENT_TYPE: application/x-www-form-urlencoded
HTTP_ACCEPT_LANGUAGE: en-US
HTTP_CONNECTION: close
HTTP_HOST: localhost
HTTP_USER_AGENT: Python-urllib/2.4
PATH_INFO: /echo.html
REQUEST_METHOD: POST
x: 1
y: 2
Body: ''
```

The body is empty because it is consumed to get form data.

We can pass a content-type explicitly:

```
>>> browser.post('http://localhost/echo.html',
...              '{"x":1,"y":2}', 'application/x-javascript')
>>> print(browser.contents)
CONTENT_LENGTH: 13
CONTENT_TYPE: application/x-javascript
HTTP_ACCEPT_LANGUAGE: en-US
HTTP_CONNECTION: close
HTTP_HOST: localhost
HTTP_USER_AGENT: Python-urllib/2.4
PATH_INFO: /echo.html
REQUEST_METHOD: POST
Body: '{"x":1,"y":2}'
```

Here, the body is left in place because it isn't form data.

Performance Testing

Browser objects keep up with how much time each request takes. This can be used to ensure a particular request's performance is within a tolerable range. Be very careful using raw seconds, cross-machine differences can be huge, pystones is usually a better choice.

```
>>> browser.open('http://localhost/@@/testbrowser/simple.html')
>>> browser.lastRequestSeconds < 10 # really big number for safety
True
>>> browser.lastRequestPystones < 10000 # really big number for safety
True
```

Handling Errors

Often WSGI middleware or the application itself gracefully handle application errors, such as invalid URLs:

```
>>> browser.open('http://localhost/invalid')
Traceback (most recent call last):
...
HTTPError: HTTP Error 404: Not Found
```

Note that the above error was thrown by `mechanize` and not by the application. For debugging purposes, however, it can be very useful to see the original exception caused by the application. In those cases you can set the `handleErrors` property of the browser to `False`. It is defaulted to `True`:

```
>>> browser.handleErrors
True
```

So when we tell the application not to handle the errors,

```
>>> browser.handleErrors = False
```

we get a different, internal error from the application:

```
>>> browser.open('http://localhost/invalid')
Traceback (most recent call last):
...
NotFound: /invalid
```

Note: Setting the `handleErrors` attribute to False will only change anything if the WSGI application obeys the `wsgi.handleErrors` or `paste.throw_errors` WSGI environment variables. i.e. it does not catch and handle the original exception when these are set appropriately.

When the testbrowser is raising HttpErrors, the errors still hit the test. Sometimes we don't want that to happen, in situations where there are edge cases that will cause the error to be predictably but infrequently raised. Time is a primary cause of this.

To get around this, one can set the raiseHttpErrors to False.

```
>>> browser.handleErrors = True
>>> browser.raiseHttpErrors = False
```

This will cause HttpErrors not to propagate.

```
>>> browser.open('http://localhost/invalid')
```

The headers are still there, though.

```
>>> '404 Not Found' in str(browser.headers)
True
```

If we don't handle the errors, and allow internal ones to propagate, however, this flag doesn't affect things.

```
>>> browser.handleErrors = False
>>> browser.open('http://localhost/invalid')
Traceback (most recent call last):
...
NotFound: /invalid

>>> browser.raiseHttpErrors = True
```

Hand-Holding

Instances of the various objects ensure that users don't set incorrect instance attributes accidentally.

```
>>> browser.nonexistant = None
Traceback (most recent call last):
...
AttributeError: 'Browser' object has no attribute 'nonexistant'

>>> form.nonexistant = None
Traceback (most recent call last):
...
AttributeError: 'Form' object has no attribute 'nonexistant'

>>> control.nonexistant = None
Traceback (most recent call last):
```

```
...
AttributeError: 'Control' object has no attribute 'nonexistant'

>>> link.nonexistant = None
Traceback (most recent call last):
...
AttributeError: 'Link' object has no attribute 'nonexistant'
```

HTTPS support

Depending on the scheme of the request the variable wsgi.url_scheme will be set correctly on the request:

```
>>> browser.open('http://localhost/echo_one.html?var=wsgi.url_scheme')
>>> print(browser.contents)
'http'

>>> browser.open('https://localhost/echo_one.html?var=wsgi.url_scheme')
>>> print(browser.contents)
'https'
```

see http://www.python.org/dev/peps/pep-3333/ for details.

CHAPTER 2

Working with Cookies

Getting started

The cookies mapping has an extended mapping interface that allows getting, setting, and deleting the cookies that the browser is remembering for the current url, or for an explicitly provided URL.

```
>>> from zope.testbrowser.ftests.wsgitestapp import WSGITestApplication
>>> from zope.testbrowser.wsgi import Browser

>>> wsgi_app = WSGITestApplication()
>>> browser = Browser(wsgi_app=wsgi_app)
```

Initially the browser does not point to a URL, and the cookies cannot be used.

```
>>> len(browser.cookies)
Traceback (most recent call last):
...
RuntimeError: no request found
>>> browser.cookies.keys()
Traceback (most recent call last):
...
RuntimeError: no request found
```

Once you send the browser to a URL, the cookies attribute can be used.

```
>>> browser.open('http://localhost/@@/testbrowser/simple.html')
>>> len(browser.cookies)
0
>>> browser.cookies.keys()
[]
>>> browser.url
'http://localhost/@@/testbrowser/simple.html'
>>> browser.cookies.url
'http://localhost/@@/testbrowser/simple.html'
>>> import zope.testbrowser.interfaces
```

```
>>> from zope.interface.verify import verifyObject
>>> verifyObject(zope.testbrowser.interfaces.ICookies, browser.cookies)
True
```

Alternatively, you can use the `forURL` method to get another instance of the cookies mapping for the given URL.

```
>>> len(browser.cookies.forURL('http://www.example.com'))
0
>>> browser.cookies.forURL('http://www.example.com').keys()
[]
>>> browser.cookies.forURL('http://www.example.com').url
'http://www.example.com'
>>> browser.url
'http://localhost/@@/testbrowser/simple.html'
>>> browser.cookies.url
'http://localhost/@@/testbrowser/simple.html'
```

Here, we use a view that will make the server set cookies with the values we provide.

```
>>> browser.open('http://localhost/set_cookie.html?name=foo&value=bar')
>>> browser.headers['set-cookie'].replace(';', '')
'foo=bar'
```

Basic Mapping Interface

Now the cookies for localhost have a value. These are examples of just the basic accessor operators and methods.

```
>>> browser.cookies['foo']
'bar'
>>> list(browser.cookies.keys())
['foo']
>>> list(browser.cookies.values())
['bar']
>>> list(browser.cookies.items())
[('foo', 'bar')]
>>> 'foo' in browser.cookies
True
>>> 'bar' in browser.cookies
False
>>> len(browser.cookies)
1
>>> print(dict(browser.cookies))
{'foo': 'bar'}
```

As you would expect, the cookies attribute can also be used to examine cookies that have already been set in a previous request. To demonstrate this, we use another view that does not set cookies but reports on the cookies it receives from the browser.

```
>>> browser.open('http://localhost/get_cookie.html')
>>> print(browser.headers.get('set-cookie'))
None
>>> browser.contents
'foo: bar'
>>> browser.cookies['foo']
'bar'
```

The standard mapping mutation methods and operators are also available, as seen here.

```
>>> browser.cookies['sha'] = 'zam'
>>> len(browser.cookies)
2
>>> import pprint
>>> pprint.pprint(sorted(browser.cookies.items()))
[('foo', 'bar'), ('sha', 'zam')]
>>> browser.open('http://localhost/get_cookie.html')
>>> print(browser.headers.get('set-cookie'))
None
>>> print(browser.contents) # server got the cookie change
foo: bar
sha: zam
>>> browser.cookies.update({'va': 'voom', 'tweedle': 'dee'})
>>> pprint.pprint(sorted(browser.cookies.items()))
[('foo', 'bar'), ('sha', 'zam'), ('tweedle', 'dee'), ('va', 'voom')]
>>> browser.open('http://localhost/get_cookie.html')
>>> print(browser.headers.get('set-cookie'))
None
>>> print(browser.contents)
foo: bar
sha: zam
tweedle: dee
va: voom
>>> del browser.cookies['foo']
>>> del browser.cookies['tweedle']
>>> browser.open('http://localhost/get_cookie.html')
>>> print(browser.contents)
sha: zam
va: voom
```

Headers

You can see the Cookies header that will be sent to the browser in the `header` attribute and the repr and str.

```
>>> browser.cookies.header
'sha=zam; va=voom'
>>> browser.cookies
<zope.testbrowser.cookies.Cookies object at ... for
 http://localhost/get_cookie.html (sha=zam; va=voom)>
>>> str(browser.cookies)
'sha=zam; va=voom'
```

Extended Mapping Interface

Read Methods: `getinfo` and `iterinfo`

`getinfo`

The `cookies` mapping also has an extended interface to get and set extra information about each cookie. `zope.testbrowser.interfaces.ICookie.getinfo()` returns a dictionary.

Here are some examples.

```
>>> browser.open('http://localhost/set_cookie.html?name=foo&value=bar')
>>> pprint.pprint(browser.cookies.getinfo('foo'))
{'comment': None,
 'commenturl': None,
 'domain': 'localhost.local',
 'expires': None,
 'name': 'foo',
 'path': '/',
 'port': None,
 'secure': False,
 'value': 'bar'}
>>> pprint.pprint(browser.cookies.getinfo('sha'))
{'comment': None,
 'commenturl': None,
 'domain': 'localhost.local',
 'expires': None,
 'name': 'sha',
 'path': '/',
 'port': None,
 'secure': False,
 'value': 'zam'}
>>> import datetime
>>> expires = datetime.datetime(2030, 1, 1, 12, 22, 33).strftime(
...     '%a, %d %b %Y %H:%M:%S GMT')
>>> browser.open(
...     'http://localhost/set_cookie.html?name=wow&value=wee&'
...     'expires=%s' %
...     (expires,))
>>> pprint.pprint(browser.cookies.getinfo('wow'))
{'comment': None,
 'commenturl': None,
 'domain': 'localhost.local',
 'expires': datetime.datetime(2030, 1, 1, 12, 22, ...tzinfo=<UTC>),
 'name': 'wow',
 'path': '/',
 'port': None,
 'secure': False,
 'value': 'wee'}
```

Max-age is converted to an “expires” value.

```
>>> browser.open(
...     'http://localhost/set_cookie.html?name=max&value=min&'
...     'max-age=3000&&comment=silly+billy')
>>> pprint.pprint(browser.cookies.getinfo('max'))
{'comment': '"silly billy"',
 'commenturl': None,
 'domain': 'localhost.local',
 'expires': datetime.datetime(..., tzinfo=<UTC>),
 'name': 'max',
 'path': '/',
 'port': None,
 'secure': False,
 'value': 'min'}
```

iterinfo

You can iterate over all of the information about the cookies for the current page using the `iterinfo` method.

```
>>> pprint.pprint(sorted(browser.cookies.iterinfo(),
...                      key=lambda info: info['name']))
...
[{'comment': None,
  'commenturl': None,
  'domain': 'localhost.local',
  'expires': None,
  'name': 'foo',
  'path': '/',
  'port': None,
  'secure': False,
  'value': 'bar'},
 {'comment': '"silly billy"',
  'commenturl': None,
  'domain': 'localhost.local',
  'expires': datetime.datetime(..., tzinfo=<UTC>),
  'name': 'max',
  'path': '/',
  'port': None,
  'secure': False,
  'value': 'min'},
 {'comment': None,
  'commenturl': None,
  'domain': 'localhost.local',
  'expires': None,
  'name': 'sha',
  'path': '/',
  'port': None,
  'secure': False,
  'value': 'zam'},
 {'comment': None,
  'commenturl': None,
  'domain': 'localhost.local',
  'expires': None,
  'name': 'va',
  'path': '/',
  'port': None,
  'secure': False,
  'value': 'voom'},
 {'comment': None,
  'commenturl': None,
  'domain': 'localhost.local',
  'expires': datetime.datetime(2030, 1, 1, 12, 22, ...tzinfo=<UTC>),
  'name': 'wow',
  'path': '/',
  'port': None,
  'secure': False,
  'value': 'wee'}]
```

Extended Examples

If you want to look at the cookies for another page, you can either navigate to the other page in the browser, or, as already mentioned, you can use the `forURL` method, which returns an ICookies instance for the new URL.

```
>>> sorted(browser.cookies.forURL(
...     'http://localhost/inner/set_cookie.html').keys())
['foo', 'max', 'sha', 'va', 'wow']
>>> extra_cookie = browser.cookies.forURL(
...     'http://localhost/inner/set_cookie.html')
>>> extra_cookie['gew'] = 'gaw'
>>> extra_cookie.getinfo('gew')['path']
'/inner'
>>> sorted(extra_cookie.keys())
['foo', 'gew', 'max', 'sha', 'va', 'wow']
>>> sorted(browser.cookies.keys())
['foo', 'max', 'sha', 'va', 'wow']

>>> browser.open('http://localhost/inner/get_cookie.html')
>>> print(browser.contents) # has gewgaw
foo: bar
gew: gaw
max: min
sha: zam
va: voom
wow: wee
>>> browser.open('http://localhost/inner/path/get_cookie.html')
>>> print(browser.contents) # has gewgaw
foo: bar
gew: gaw
max: min
sha: zam
va: voom
wow: wee
>>> browser.open('http://localhost/get_cookie.html')
>>> print(browser.contents) # NO gewgaw
foo: bar
max: min
sha: zam
va: voom
wow: wee
```

Here's an example of the server setting a cookie that is only available on an inner page.

```
>>> browser.open(
...     'http://localhost/inner/path/set_cookie.html?name=big&value=kahuna'
...     )
>>> browser.cookies['big']
'kahuna'
>>> browser.cookies.getinfo('big')['path']
'/inner/path'
>>> browser.cookies.getinfo('gew')['path']
'/inner'
>>> browser.cookies.getinfo('foo')['path']
'/'
>>> print(browser.cookies.forURL('http://localhost/').get('big'))
None
```

Write Methods: `create` and `change`

The basic mapping API only allows setting values. If a cookie already exists for the given name, it's value will be changed; or else a new cookie will be created for the current request's domain and a path of '/', set to last for only this browser session (a "session" cookie).

To create or change cookies with different additional information, use the `create` and `change` methods, respectively. Here is an example of `create`.

```
>>> from pytz import UTC
>>> browser.cookies.create(
...     'bling', value='blang', path='/inner',
...     expires=datetime.datetime(2020, 1, 1, tzinfo=UTC),
...     comment='follow swallow')
>>> pprint.pprint(browser.cookies.getinfo('bling'))
{'comment': 'follow%20swallow',
 'commenturl': None,
 'domain': 'localhost.local',
 'expires': datetime.datetime(2020, 1, 1, 0, 0, tzinfo=<UTC>),
 'name': 'bling',
 'path': '/inner',
 'port': None,
 'secure': False,
 'value': 'blang'}
```

In these further examples of `create`, note that the testbrowser sends all domains to Zope, and both http and https.

```
>>> browser.open('https://dev.example.com/inner/path/get_cookie.html')
>>> browser.cookies.keys() # a different domain
[]
>>> browser.cookies.create('tweedle', 'dee')
>>> pprint.pprint(browser.cookies.getinfo('tweedle'))
{'comment': None,
 'commenturl': None,
 'domain': 'dev.example.com',
 'expires': None,
 'name': 'tweedle',
 'path': '/inner/path',
 'port': None,
 'secure': False,
 'value': 'dee'}
>>> browser.cookies.create(
...     'boo', 'yah', domain='.example.com', path='/inner', secure=True)
>>> pprint.pprint(browser.cookies.getinfo('boo'))
{'comment': None,
 'commenturl': None,
 'domain': '.example.com',
 'expires': None,
 'name': 'boo',
 'path': '/inner',
 'port': None,
 'secure': True,
 'value': 'yah'}
>>> sorted(browser.cookies.keys())
['boo', 'tweedle']
>>> browser.open('https://dev.example.com/inner/path/get_cookie.html')
>>> print(browser.contents)
boo: yah
tweedle: dee
```

```
>>> browser.open( # not https, so not secure, so not 'boo'
...      'http://dev.example.com/inner/path/get_cookie.html')
>>> sorted(browser.cookies.keys())
['tweedle']
>>> print(browser.contents)
tweedle: dee
>>> browser.open( # not tweedle's domain
...      'https://prod.example.com/inner/path/get_cookie.html')
>>> sorted(browser.cookies.keys())
['boo']
>>> print(browser.contents)
boo: yah
>>> browser.open( # not tweedle's domain
...      'https://example.com/inner/path/get_cookie.html')
>>> sorted(browser.cookies.keys())
['boo']
>>> print(browser.contents)
boo: yah
>>> browser.open( # not tweedle's path
...      'https://dev.example.com/inner/get_cookie.html')
>>> sorted(browser.cookies.keys())
['boo']
>>> print(browser.contents)
boo: yah
```

Masking by Path

The API allows creation of cookies that mask existing cookies, but it does not allow creating a cookie that will be immediately masked upon creation. Having multiple cookies with the same name for a given URL is rare, and is a pathological case for using a mapping API to work with cookies, but it is supported to some degree, as demonstrated below. Note that the Cookie RFCs (2109, 2965) specify that all matching cookies be sent to the server, but with an ordering so that more specific paths come first. We also prefer more specific domains, though the RFCs state that the ordering of cookies with the same path is indeterminate. The best-matching cookie is the one that the mapping API uses.

Also note that ports, as sent by RFC 2965's Cookie2 and Set-Cookie2 headers, are parsed and stored by this API but are not used for filtering as of this writing.

This is an example of making one cookie that masks another because of path. First, unless you pass an explicit path, you will be modifying the existing cookie.

```
>>> browser.open('https://dev.example.com/inner/path/get_cookie.html')
>>> print(browser.contents)
boo: yah
tweedle: dee
>>> browser.cookies.getinfo('boo')['path']
'/inner'
>>> browser.cookies['boo'] = 'hoo'
>>> browser.cookies.getinfo('boo')['path']
'/inner'
>>> browser.cookies.getinfo('boo')['secure']
True
```

Now we mask the cookie, using the path.

```
>>> browser.cookies.create('boo', 'boo', path='/inner/path')
>>> browser.cookies['boo']
'boo'
>>> browser.cookies.getinfo('boo')['path']
'/inner/path'
>>> browser.cookies.getinfo('boo')['secure']
False
>>> browser.cookies['boo']
'boo'
>>> sorted(browser.cookies.keys())
['boo', 'tweedle']
```

To identify the additional cookies, you can change the URL...

```
>>> extra_cookies = browser.cookies.forURL(
...     'https://dev.example.com/inner/get_cookie.html')
>>> extra_cookies['boo']
'hoo'
>>> extra_cookies.getinfo('boo')['path']
'/inner'
>>> extra_cookies.getinfo('boo')['secure']
True
```

...or use `iterinfo` and pass in a name.

```
>>> pprint.pprint(list(browser.cookies.iterinfo('boo')))
[{'comment': None,
  'commenturl': None,
  'domain': 'dev.example.com',
  'expires': None,
  'name': 'boo',
  'path': '/inner/path',
  'port': None,
  'secure': False,
  'value': 'boo'},
 {'comment': None,
  'commenturl': None,
  'domain': '.example.com',
  'expires': None,
  'name': 'boo',
  'path': '/inner',
  'port': None,
  'secure': True,
  'value': 'hoo'}]
```

An odd situation in this case is that deleting a cookie can sometimes reveal another one.

```
>>> browser.open('https://dev.example.com/inner/path/get_cookie.html')
>>> browser.cookies['boo']
'boo'
>>> del browser.cookies['boo']
>>> browser.cookies['boo']
'hoo'
```

Creating a cookie that will be immediately masked within the current url is not allowed.

```
>>> browser.cookies.getinfo('tweedle')['path']
'/inner/path'
```

```
>>> browser.cookies.create('tweedle', 'dum', path='/inner')
...
Traceback (most recent call last):
...
ValueError: cannot set a cookie that will be hidden by another cookie for
this url (https://dev.example.com/inner/path/get_cookie.html)
>>> browser.cookies['tweedle']
'dee'
```

Masking by Domain

All of the same behavior is also true for domains. The only difference is a theoretical one: while the behavior of masking cookies via paths is defined by the relevant IRCs, it is not defined for domains. Here, we simply follow a "best match" policy.

We initialize by setting some cookies for example.org.

```
>>> browser.open('https://dev.example.org/get_cookie.html')
>>> browser.cookies.keys() # a different domain
[]
>>> browser.cookies.create('tweedle', 'dee')
>>> browser.cookies.create('boo', 'yah', domain='example.org',
...                        secure=True)
```

Before we look at the examples, note that the default behavior of the cookies is to be liberal in the matching of domains.

```
>>> browser.cookies.strict_domain_policy
False
```

According to the RFCs, a domain of 'example.com' can only be set implicitly from the server, and implies an exact match, so example.com URLs will get the cookie, but not `*.example.com` (i.e., `dev.example.com`). Real browsers vary in their behavior in this regard. The cookies collection, by default, has a looser interpretation of this, such that domains are always interpreted as effectively beginning with a ".", so `dev.example.com` will include a cookie from the `example.com` domain filter as if it were a `.example.com` filter.

Here's an example. If we go to `dev.example.org`, we should only see the "tweedle" cookie if we are using strict rules. But right now we are using loose rules, so 'boo' is around too.

```
>>> browser.open('https://dev.example.org/get_cookie.html')
>>> sorted(browser.cookies)
['boo', 'tweedle']
>>> print(browser.contents)
boo: yah
tweedle: dee
```

If we set `strict_domain_policy` to True, then only tweedle is included.

```
>>> browser.cookies.strict_domain_policy = True
>>> sorted(browser.cookies)
['tweedle']
>>> browser.open('https://dev.example.org/get_cookie.html')
>>> print(browser.contents)
tweedle: dee
```

If we set the "boo" domain to `.example.org` (as it would be set under the more recent Cookie RFC if a server sent the value) then maybe we get the "boo" value again.

```
>>> browser.cookies.forURL('https://example.org').change(
...     'boo', domain=".example.org")
Traceback (most recent call last):
...
ValueError: policy does not allow this cookie
```

Whoa! Why couldn't we do that?

Well, the strict_domain_policy affects what cookies we can set also. With strict rules, ".example.org" can only be set by "*.example.org" domains, *not* example.org itself.

OK, we'll create a new cookie then.

```
>>> browser.cookies.forURL('https://snoo.example.org').create(
...     'snoo', 'kums', domain=".example.org")

>>> sorted(browser.cookies)
['snoo', 'tweedle']
>>> browser.open('https://dev.example.org/get_cookie.html')
>>> print(browser.contents)
snoo: kums
tweedle: dee
```

Let's set things back to the way they were.

```
>>> del browser.cookies['snoo']
>>> browser.cookies.strict_domain_policy = False
>>> browser.open('https://dev.example.org/get_cookie.html')
>>> sorted(browser.cookies)
['boo', 'tweedle']
>>> print(browser.contents)
boo: yah
tweedle: dee
```

Now back to the the examples of masking by domain. First, unless you pass an explicit domain, you will be modifying the existing cookie.

```
>>> browser.cookies.getinfo('boo')['domain']
'example.org'
>>> browser.cookies['boo'] = 'hoo'
>>> browser.cookies.getinfo('boo')['domain']
'example.org'
>>> browser.cookies.getinfo('boo')['secure']
True
```

Now we mask the cookie, using the domain.

```
>>> browser.cookies.create('boo', 'boo', domain='dev.example.org')
>>> browser.cookies['boo']
'boo'
>>> browser.cookies.getinfo('boo')['domain']
'dev.example.org'
>>> browser.cookies.getinfo('boo')['secure']
False
>>> browser.cookies['boo']
'boo'
>>> sorted(browser.cookies.keys())
['boo', 'tweedle']
```

To identify the additional cookies, you can change the URL...

```
>>> extra_cookies = browser.cookies.forURL(
...     'https://example.org/get_cookie.html')
>>> extra_cookies['boo']
'hoo'
>>> extra_cookies.getinfo('boo')['domain']
'example.org'
>>> extra_cookies.getinfo('boo')['secure']
True
```

...or use `iterinfo` and pass in a name.

```
>>> pprint.pprint(list(browser.cookies.iterinfo('boo')))
[{'comment': None,
  'commenturl': None,
  'domain': 'dev.example.org',
  'expires': None,
  'name': 'boo',
  'path': '/',
  'port': None,
  'secure': False,
  'value': 'boo'},
 {'comment': None,
  'commenturl': None,
  'domain': 'example.org',
  'expires': None,
  'name': 'boo',
  'path': '/',
  'port': None,
  'secure': True,
  'value': 'hoo'}]
```

An odd situation in this case is that deleting a cookie can sometimes reveal another one.

```
>>> browser.open('https://dev.example.org/get_cookie.html')
>>> browser.cookies['boo']
'boo'
>>> del browser.cookies['boo']
>>> browser.cookies['boo']
'hoo'
```

Setting a cookie with a foreign domain from the current URL is not allowed (use forURL to get around this).

```
>>> browser.cookies.create('tweedle', 'dum', domain='locahost.local')
Traceback (most recent call last):
...
ValueError: current url must match given domain
>>> browser.cookies['tweedle']
'dee'
```

Setting a cookie that will be immediately masked within the current url is also not allowed.

```
>>> browser.cookies.getinfo('tweedle')['domain']
'dev.example.org'
>>> browser.cookies.create('tweedle', 'dum', domain='.example.org')
...
Traceback (most recent call last):
...
```

```
ValueError: cannot set a cookie that will be hidden by another cookie for
this url (https://dev.example.org/get_cookie.html)
>>> browser.cookies['tweedle']
'dee'
```

change

So far all of our examples in this section have centered on `create`. `change` allows making changes to existing cookies. Changing expiration is a good example.

```
>>> browser.open("http://localhost/@@/testbrowser/cookies.html")
>>> browser.cookies['foo'] = 'bar'
>>> browser.cookies.change('foo', expires=datetime.datetime(2021, 1, 1))
>>> browser.cookies.getinfo('foo')['expires']
datetime.datetime(2021, 1, 1, 0, 0, tzinfo=<UTC>)
```

That's the main story. Now here are some edge cases.

```
>>> browser.cookies.change(
...     'foo',
...     expires=zope.testbrowser.cookies.expiration_string(
...         datetime.datetime(2020, 1, 1)))
>>> browser.cookies.getinfo('foo')['expires']
datetime.datetime(2020, 1, 1, 0, 0, tzinfo=<UTC>)

>>> browser.cookies.forURL(
...   'http://localhost/@@/testbrowser/cookies.html').change(
...     'foo',
...     expires=zope.testbrowser.cookies.expiration_string(
...         datetime.datetime(2019, 1, 1)))
>>> browser.cookies.getinfo('foo')['expires']
datetime.datetime(2019, 1, 1, 0, 0, tzinfo=<UTC>)
>>> browser.cookies['foo']
'bar'
>>> browser.cookies.change('foo', expires=datetime.datetime(1999, 1, 1))
>>> len(browser.cookies)
4
```

While we are at it, it is worth noting that trying to create a cookie that has already expired raises an error.

```
>>> browser.cookies.create('foo', 'bar',
...                        expires=datetime.datetime(1999, 1, 1))
Traceback (most recent call last):
...
AlreadyExpiredError: May not create a cookie that is immediately expired
```

Clearing cookies

clear, clearAll, clearAllSession allow various clears of the cookies.

The `clear` method clears all of the cookies for the current page.

```
>>> browser.open('http://localhost/@@/testbrowser/cookies.html')
>>> len(browser.cookies)
4
```

```
>>> browser.cookies.clear()
>>> len(browser.cookies)
0
```

The `clearAllSession` method clears *all* session cookies (for all domains and paths, not just the current URL), as if the browser had been restarted.

```
>>> browser.cookies.clearAllSession()
>>> len(browser.cookies)
0
```

The `clearAll` removes all cookies for the browser.

```
>>> browser.cookies.clearAll()
>>> len(browser.cookies)
0
```

Note that explicitly setting a Cookie header is an error if the `cookies` mapping has any values; and adding a new cookie to the `cookies` mapping is an error if the Cookie header is already set. This is to prevent hard-to- diagnose intermittent errors when one header or the other wins.

```
>>> browser.cookies['boo'] = 'yah'
>>> browser.addHeader('Cookie', 'gee=gaw')
Traceback (most recent call last):
...
ValueError: cookies are already set in `cookies` attribute
```

```
>>> browser.cookies.clearAll()
>>> browser.addHeader('Cookie', 'gee=gaw')
>>> browser.cookies['fee'] = 'fi'
Traceback (most recent call last):
...
ValueError: cookies are already set in `Cookie` header
```

CHAPTER 3

zope.testbrowser API

zope.testbrowser.interfaces

Interfaces

Exceptions

CHAPTER 4

Indices and tables

- genindex
- modindex
- search