Reference

adt Module

High level abstract datatypes

class webscraping.adt.**Bag**(**args*, ***kwargs*) [source]

Bases: **dict**

Dictionary object with attribute like access

```
>>> b = Bag()
>>> b.name = 'company'
>>> b.name
'company'
>>> b.address
```

class webscraping.adt.**HashDict**(*default_factory=<type 'str'>*) [source]

For storing large quantities of keys where don’t need the original value of the key Instead each key is hashed and hashes are compared for equality

```
>>> hd = HashDict()
>>> url = 'http://webscraping.com'
>>> hd[url] = True
>>> url in hd
True
>>> 'other url' in hd
False
>>> len(hd)
1
```

get(*name*, *default=None*) [source]

Get the value at this key

Returns default if key does not exist

get_hash(*value*) [source]

get the hash value of this value

alg Module

High level functions for interpreting useful data from input

webscraping.alg.**chrome_browser**(*os_version*) [source]

webscraping.alg.**distance**(*p1*, *p2*, *scale=None*) [source]

Calculate distance between 2 (latitude, longitude) points.

scale:
: By default the distance will be returned as a ratio of the earth's radius Use 'km' to return distance in kilometres, 'miles' to return distance in miles

```
>>> melbourne = -37.7833, 144.9667
>>> san_francisco = 37.7750, -122.4183
>>> int(distance(melbourne, san_francisco, 'km'))
12659
```

`webscraping.alg.`**`extract_emails`**(*html*) [source]

Remove common obfuscations from HTML and then extract all emails

```
>>> extract_emails('')
[]
>>> extract_emails('hello contact@webscraping.com world')
['contact@webscraping.com']
>>> extract_emails('hello contact@<!-- trick comment -->webscraping.com world')
['contact@webscraping.com']
>>> extract_emails('hello contact AT webscraping DOT com world')
['contact@webscraping.com']
```

`webscraping.alg.`**`extract_phones`**(*html*) [source]

Extract phone numbers from this HTML

```
>>> extract_phones('Phone: (123) 456-7890 <br>')
['(123) 456-7890']
>>> extract_phones('Phone 123.456.7890 ')
['123.456.7890']
>>> extract_phones('+1-123-456-7890<br />123 456 7890n')
['+1-123-456-7890', '123 456 7890']
>>> extract_phones('456-7890')
[]
```

`webscraping.alg.`**`firefox_browser`**(*os_version*) [source]

`webscraping.alg.`**`get_excerpt`**(*html*, *try_meta=False*, *max_chars=255*) [source]

Extract excerpt from this HTML by finding the largest text block

try_meta:
: indicates whether to try extracting from meta description tag

max_chars:
: the maximum number of characters for the excerpt

`webscraping.alg.`**`ie_browser`**(*os_version=None*) [source]

`webscraping.alg.`**`linux_os`**() [source]

webscraping.alg.**osx_os**() [source]

webscraping.alg.**parse_us_address**(*address*) [source]

Parse USA address into address, city, state, and zip code

```
>>> parse_us_address('6200 20th Street, Vero Beach, FL 32966')
('6200 20th Street', 'Vero Beach', 'FL', '32966')
```

webscraping.alg.**rand_agent**() [source]

Returns a random user agent across Firefox, IE, and Chrome on Linux, OSX, and Windows

webscraping.alg.**rand_os**() [source]

webscraping.alg.**windows_os**() [source]

common Module

Common web scraping related functions

class webscraping.common.**ConsoleHandler** [source]

Bases: **logging.StreamHandler**

Log to stderr for errors else stdout

emit(*record*) [source]

class webscraping.common.**UnicodeWriter**(*file*, *encoding='utf-8'*, *mode='wb'*, *unique=False*, *unique_by=None*, *quoting=1*, *utf8_bom=False*, *auto_repair=False*, ***argv*) [source]

A CSV writer that produces Excel-compatible CSV files from unicode data.

file:
: can either be a filename or a file object

encoding:
: the encoding to use for output

mode:
: the mode for writing to file

unique:
: if True then will only write unique rows to output

unique_by:
: make the rows unique by these columns(the value is a list of indexs), default by all columns

quoting:
: csv module quoting style to use

utf8_bom:
: whether need to add the BOM

auto_repair:
: whether need to remove the invalid rows automatically

```
>>> from StringIO import StringIO
>>> fp = StringIO()
>>> writer = UnicodeWriter(fp, quoting=csv.QUOTE_MINIMAL)
>>> writer.writerow(['a', '1'])
>>> writer.flush()
>>> fp.seek(0)
>>> fp.read().strip()
'a,1'
```

close() [source]

Close the output file pointer

flush() [source]

Flush output to disk

writerow(*row*) [source]

Write row to output

writerows(*rows*) [source]

Write multiple rows to output

exception webscraping.common.**WebScrapingError** [source]

Bases: **exceptions.Exception**

webscraping.common.**firefox_cookie**(*file=None*, *tmp_sqlite_file='cookies.sqlite'*, *tmp_cookie_file='cookies.txt'*) [source]

Create a cookie jar from this FireFox 3 sqlite cookie database

```
>>> cj = firefox_cookie()
>>> opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cj))
>>> url = 'http://code.google.com/p/webscraping'
>>> html = opener.open(url).read()
```

webscraping.common.**first**(*l*, *default=''*) [source]

Return first element from list or default value if out of range

```
>>> first([1,2,3])
1
>>> first([], None)
```

webscraping.common.**get_domain**(*url*) [source]

Extract the domain from the given URL

```
>>> get_domain('http://www.google.com.au/tos.html')
'google.com.au'
>>> get_domain('www.google.com')
'google.com'
```

webscraping.common.**get_extension**(*url*) [source]

Return extension from given URL

```
>>> get_extension('hello_world.JPG')
'jpg'
>>> get_extension('http://www.google-analytics.com/__utm.gif?utmwv=1.3&utmn=420639071'
'gif'
```

webscraping.common.**get_logger**(*output_file*, *level=20*, *maxbytes=0*) [source]

Create a logger instance

output_file:
: file where to save the log

level:
: the minimum logging level to save

maxbytes:
: the maxbytes allowed for the log file size. 0 means no limit.

webscraping.common.**html_to_unicode**(*html*, *charset='utf-8'*) [source]

Convert html to unicode, decoding by specified charset when available

webscraping.common.**is_html**(*html*) [source]

Returns whether content is likely HTML based on search for common tags

webscraping.common.**last**(*l*, *default=''*) [source]

Return last element from list or default value if out of range

webscraping.common.**normalize**(*s*, *encoding='utf-8'*) [source]

Normalize the string by removing tags, unescaping, and removing surrounding whitespace

```
>>> normalize('''<span>Tel.:   029&nbsp;-&nbsp;12345678   </span>''')
'Tel.: 029 - 12345678'
```

webscraping.common.**nth**(*l*, *i*, *default=''*) [source]

Return nth item from list or default value if out of range webscraping.common.**pad**(*l*, *size*, *default=None*, *end=True*) [source]

Return list of given size Insert elements of default value if too small Remove elements if too large Manipulate end of list if end is True, else start

```
>>> pad(range(5), 5)
[0, 1, 2, 3, 4]
>>> pad(range(5), 3)
[0, 1, 2]
>>> pad(range(5), 7, -1)
[0, 1, 2, 3, 4, -1, -1]
>>> pad(range(5), 7, end=False)
[None, None, 0, 1, 2, 3, 4]
```

webscraping.common.**parse_proxy**(*proxy*) [source]

Parse a proxy into its fragments Returns a dict with username, password, host, and port

```
>>> f = parse_proxy('login:pw@66.197.208.200:8080')
>>> f.username
'login'
>>> f.password
'pw'
>>> f.host
'66.197.208.200'
>>> f.port
'8080'
>>> f = parse_proxy('66.197.208.200')
>>> f.username == f.password == f.port == ''
True
>>> f.host
'66.197.208.200'
```

webscraping.common.**pretty**(*s*) [source]

Return pretty version of string for display

```
>>> pretty('hello_world')
'Hello World'
```

webscraping.common.**pretty_duration**(*dt*) [source]

Return english description of this time difference

```
>>> from datetime import timedelta
>>> pretty_duration(timedelta(seconds=1))
'1 second'
>>> pretty_duration(timedelta(hours=1))
'1 hour'
>>> pretty_duration(timedelta(days=2))
'2 days'
```

webscraping.common.**pretty_paragraph**(*s*) [source]

Return pretty version of text in paragraph for display webscraping.common.**read_list**(*file*) [source]

Return file as list if exists

webscraping.common.**regex_get**(*html*, *pattern*, *index=None*, *normalized=True*, *flag=18*, *default=''*) [source]

Helper method to extract content from regular expression

```
>>> regex_get('<div><span>Phone: 029&nbsp;01054609</span><span></span></div>', r'<span>
'029 01054609'
>>> regex_get('<div><span>Phone: 029&nbsp;01054609</span><span></span></div>', r'<span>
['029', '01054609']
```

webscraping.common.**remove_tags**(*html*, *keep_children=True*) [source]

Remove HTML tags leaving just text If keep children is True then keep text within child tags

```
>>> remove_tags('hello <b>world</b>!')
'hello world!'
>>> remove_tags('hello <b>world</b>!', False)
'hello !'
>>> remove_tags('hello <br>world<br />!', False)
'hello world!'
>>> remove_tags('<span><b></b></span>test</span>', False)
'test'
```

webscraping.common.**safe**(*s*) [source]

Return characters in string that are safe for URLs

```
>>> safe('U@#$_#^&*-2')
'U_-2'
```

webscraping.common.**same_domain**(*url1*, *url2*) [source]

Return whether URLs belong to same domain

```
>>> same_domain('http://www.google.com.au', 'code.google.com')
True
>>> same_domain('http://www.facebook.com', 'http://www.myspace.com')
False
```

webscraping.common.**start_threads**(*fn*, *num_threads=20*, *args=()*, *wait=True*) [source]

Shortcut to start these threads with given args and wait for all to finish

webscraping.common.**to_ascii**(*html*) [source]

Return ascii part of html

webscraping.common.**to_float**(*s*, *default=0.0*) [source]

Return float from this string

```
>>> to_float('90.45')
90.45
>>> to_float('')
0.0
>>> to_float('90')
90.0
>>> to_float('..9')
0.0
>>> to_float('.9')
0.9
>>> to_float(None)
0.0
>>> to_float(1)
1.0
```

`webscraping.common.`**`to_int`**(*s*, *default=0*) [source]

Return integer from this string

```
>>> to_int('90')
90
>>> to_int('-90.2432')
-90
>>> to_int('a90a')
90
>>> to_int('a')
0
>>> to_int('a', 90)
90
```

`webscraping.common.`**`to_unicode`**(*obj*, *encoding='utf-8'*) [source]

Convert obj to unicode

`webscraping.common.`**`unescape`**(*text*, *encoding='utf-8'*, *keep_unicode=False*) [source]

Interpret escape characters

```
>>> unescape('&lt;hello&nbsp;&amp;%20world&gt;')
'<hello & world>'
```

`webscraping.common.`**`unique`**(*l*) [source]

Remove duplicates from list, while maintaining order

```
>>> unique([3,6,4,4,6])
[3, 6, 4]
>>> unique([])
[]
>>> unique([3,6,4])
[3, 6, 4]
```

`download` Module

Helper methods to download and crawl web content using threads *class* `webscraping.download.`**`CrawlerCallback`**(*output_file=None*, *max_links=100*, *max_depth=1*, *allowed_urls=''*, *banned_urls='^$'*, *robots=None*, *crawl_existing=True*) [source]

Example callback to crawl a website

`crawl`(*D*, *url*, *html*) [source]

Crawl website html and return list of URLs crawled

`found` = *<webscraping.adt.HashDict instance at 0x377c050>*

class `webscraping.download.`**`Download`**(*cache=None*, *cache_file=None*, *read_cache=True*, *write_cache=True*, *use_network=True*, *user_agent=None*, *timeout=30*, *delay=5*, *proxies=None*, *proxy_file=None*, *max_proxy_errors=5*, *opener=None*, *headers=None*, *data=None*, *num_retries=0*, *num_redirects=0*, *force_html=False*, *force_ascii=False*, *max_size=None*, *default=''*, *pattern=None*, *acceptable_errors=None*, ***kwargs*) [source]

cache:
: a pdict object to use for the cache

cache_file:
: filename to store cached data

read_cache:
: whether to read from the cache

write_cache:
: whether to write to the cache

use_network:
: whether to download content not in the cache

user_agent
: the User Agent to download content with

timeout:
: the maximum amount of time to wait for http response

delay:
: the minimum amount of time (in seconds) to wait after downloading content from a domain per proxy

proxy_file:
: a filename to read proxies from

max_proxy_errors:
: the maximum number of consecutive errors allowed per proxy before discarding an error is only counted if another proxy is able to successfully download the URL set to None to disable

proxies:
: a list of proxies to cycle through when downloading content

opener:
: an optional opener to use instead of using urllib2 directly

headers:
: the headers to include in the request

data:
: what to post at the URL if None (default) then a GET request will be made

num_retries:
: how many times to try downloading a URL when get an error

num_redirects:
: how many times the URL is allowed to be redirected, to avoid infinite loop

force_html:
: whether to download non-text data

force_ascii:
: whether to only return ascii characters

max_size:
: maximum number of bytes that will be downloaded, or None to disable

default:
: what to return when no content can be downloaded

pattern:
: a regular expression that the downloaded HTML has to match to be considered a valid download

acceptable_errors:
: a list contains all acceptable HTTP codes, don't try downloading for them e.g. no need to retry for 404 error

archive_get(*url*, *timestamp=None*, ***kwargs*) [source]

Download webpage via the archive.org cache

url:
: The webpage to download

timestamp:
: When passed a datetime object will download the cached webpage closest to this date, Else when None (default) will download the most recent archived page.

exists(*url*) [source]

Do a HEAD request to check whether webpage exists

fetch(*url*, *headers=None*, *data=None*, *proxy=None*, *user_agent=None*, *opener=None*, *pattern=None*) [source]

Simply download the url and return the content

gcache_get(*url*, ***kwargs*) [source]

Download webpage via google cache

geocode(*address*, *delay=5*, *read_cache=True*, *num_retries=1*, *language=None*) [source]

get(*url*, ***kwargs*) [source]

Download this URL and return the HTML. By default HTML is cached so only have to download once.

url:
: what to download

kwargs:
: override any of the arguments passed to constructor

get_emails(*website*, *max_depth=1*, *max_urls=10*, *max_emails=1*) [source]

Crawl this website and return all emails found

website:
: the URL of website to crawl

max_depth:
: how many links deep to follow before stop crawl

max_urls:
: how many URL's to download before stop crawl

max_emails:
: The maximum number of emails to extract before stop crawl. If None then extract all emails found in crawl.

get_key(*url*, *data=None*) [source]

Create key for caching this request

get_proxy(*proxies=None*) [source]

Return random proxy if available

get_user_agent(*proxy*) [source]

Get user agent for this proxy

gtrans_get(*url*, ***kwargs*) [source]

Download webpage via Google Translation

invalid_response(*html*, *pattern*) [source]

Return whether the response contains a regex error pattern

places(*api_key*, *keyword*, *latitude*, *longitude*, *radius=10000*, *delay=5*, *num_retries=1*, *language='en'*) [source]

proxy_agents = *{}*

proxy_performance = *<webscraping.download.ProxyPerformance instance at 0x3771ea8>*

reload_proxies(*timeout=600*) [source]

Check periodically for updated proxy file

timeout:
: the number of seconds before check for updated proxies

save_as(*url*, *filename=None*, *save_dir='images'*) [source]

Download url and save to disk

url:
: the webpage to download

filename:
: Output file to save to. If not set then will save to file based on URL

throttle(*url*, *delay*, *proxy=None*, *variance=0.5*) [source]

Delay a minimum time for each domain per proxy by storing last access time

url
: what intend to download

delay
: the minimum amount of time (in seconds) to wait after downloading content from this domain

proxy
: the proxy to download through

variance
: the amount of randomness in delay, 0-1

whois(*url*, *timeout=10*) [source]

Return text of this whois query

class `webscraping.download.`**`GoogleMaps`**(*D*) [source]

`geocode`(*address*, *delay=5*, *read_cache=True*, *num_retries=1*, *language=None*) [source]

Geocode address using Google's API and return dictionary of useful fields

address:
    what to pass to geocode API

delay:
    how long to delay between API requests

read_cache:
    whether to load content from cache when exists

num_retries:
    the number of times to try downloading

language:
    the language to set

`load_result`(*url*, *html*) [source]

Parse the result from API

If JSON is well formed and status is OK then will return result Else will return an empty dict

`parse_location`(*result*) [source]

Parse address data from Google's geocoding response into a more usable flat structure

Example: https://developers.google.com/maps/documentation/geocoding/#JSON

`places`(*api_key*, *keyword*, *latitude*, *longitude*, *radius=10000*, *delay=5*, *num_retries=1*, *language='en'*) [source]

Search the Google Place API for this keyword and location

api_key is the Google Places API key: https://developers.google.com/places/documentation/#Authentication radius around the location can be a maximum 50000

Returns a list of up to 200 matching places

class `webscraping.download.`**`ProxyPerformance`** [source]

Track performance of proxies If 10 errors in a row that other proxies could handle then need to remove

error(*proxy*) [source]

Add to error count and returns number of consecutive errors for this proxy

success(*proxy*) [source]

Successful download - so clear error count

class `webscraping.download.`**State**(*output_file=None*, *timeout=10*) [source]

Save state of crawl to disk

output_file:
where to save the state

timeout:
how many seconds to wait between saving the state

save() [source]

Save state to disk

update(*num_downloads=0*, *num_errors=0*, *num_caches=0*, *queue_size=0*) [source]

Update the state with these values

num_downloads:
the number of downloads completed successfully

num_errors:
the number of errors encountered while downloading

num_caches:
the number of webpages read from cache instead of downloading

queue_size:
the number of URL's in the queue

exception `webscraping.download.`**StopCrawl** [source]

Bases: **exceptions.Exception**

Raise this exception to interrupt crawl

`webscraping.download.`**get_redirect**(*url*, *html*) [source]

Check for meta redirects and return redirect URL if found

`webscraping.download.`**threaded_get**(*url=None*, *urls=None*, *url_iter=None*, *num_threads=10*, *dl=None*, *cb=None*, *depth=None*, *wait_finish=True*, *reuse_queue=False*, *max_queue=1000*, ***kwargs*) [source]

Download these urls in parallel

url:

the webpage to download

urls:
: the webpages to download

num_threads:
: the number of threads to download urls with

cb:
: Called after each download with the HTML of the download. The arguments are the url and downloaded html. Whatever URLs are returned are added to the crawl queue.

dl:
: A callback for customizing the download. Takes the download object and url and should return the HTML.

depth:
: True for depth first search

wait_finish:
: whether to wait until all download threads have finished before returning

reuse_queue:
: Whether to continue the queue from the previous run.

max_queue:
: The maximum number of queued URLs to keep in memory. The rest will be in the cache.

`pdict` Module

pdict has a dictionary like interface and a sqlite backend It uses pickle to store Python objects and strings, which are then compressed Multithreading is supported

class `webscraping.pdict.`**`FSCache`**(*folder*) [source]

Dictionary interface that stores cached values in the file system rather than in memory. The file path is formed from an md5 hash of the key.

folder:
: the root level folder for the cache

```
>>> fscache = FSCache('.')
>>> url = 'http://google.com/abc'
>>> html = '<html>abc</html>'
>>> url in fscache
False
>>> fscache[url] = html
>>> url in fscache
```

```
True
>>> fscache.get(url) == html
True
>>> fscache.get(html) == ''
True
>>> fscache.clear()
```

FILE_NAME = *'index.html'*

PARENT_DIR = *'fscache'*

clear() [source]

Remove all the cached values

get(*key*, *default=''*) [source]

Get data at this key and return default if does not exist

class webscraping.pdict.**PersistentDict**(*filename='cache.db'*, *compress_level=6*, *expires=None*, *timeout=10000*, *isolation_level=None*) [source]

Stores and retrieves persistent data through a dict-like interface Data is stored compressed on disk using sqlite3

filename:

where to store sqlite database. Uses in memory by default.

compress_level:

between 1-9 (in my test levels 1-3 produced a 1300kb file in ~7 seconds while 4-9 a 288kb file in ~9 seconds)

expires:

a timedelta object of how old data can be before expires. By default is set to None to disable.

timeout:

how long should a thread wait for sqlite to be ready (in ms)

isolation_level:

None for autocommit or else 'DEFERRED' / 'IMMEDIATE' / 'EXCLUSIVE'

```
>>> filename = 'cache.db'
>>> cache = PersistentDict(filename)
>>> url = 'http://google.com/abc'
>>> html = '<html>abc</html>'
>>>
>>> url in cache
False
>>> cache[url] = html
>>> url in cache
True
>>> cache[url] == html
True
```

```
>>> cache.get(url)['value'] == html
True
>>> now = datetime.datetime.now()
>>> cache.meta(url)
{}
>>> cache.meta(url, 'meta')
>>> cache.meta(url)
'meta'
>>> del cache[url]
>>> url in cache
False
>>> key, value = 'language', 'python'
>>> cache.update([(url, value), (key, value)])
>>> cache[url] == value
True
>>> cache[key] == value
True
>>> os.remove(filename)
```

clear() [source]

Clear all cached data

deserialize(*value*) [source]

convert compressed pickled string from database back into an object

get(*key*, *default=None*) [source]

Get data at key and return default if not defined

is_fresh(*t*) [source]

returns whether this datetime has expired

merge(*db*, *override=False*) [source]

Merge this databases content override determines whether to override existing keys

meta(*key*, *value=None*) [source]

Get / set meta for this value

if value is passed then set the meta attribute for this key if not then get the existing meta data for this key

serialize(*value*) [source]

convert object to a compressed pickled string to save in the db

update(*key_values*) [source]

Add this list of (key, value) tuples in single transaction to database

class webscraping.pdict.**Queue**(*filename*, *timeout=10000*, *isolation_level=None*) [source]

Stores queue of outstanding URL's on disk

```
>>> filename = 'queue.db'
>>> queue = Queue(filename)
>>> keys = [('a', 1), ('b', 2), ('c', 1)]
>>> queue.push(keys) # add new keys
>>> len(queue)
3
>>> queue.push(keys) # trying adding duplicate keys
>>> len(queue)
3
>>> queue.clear(keys=['a'])
1
>>> queue.pull(limit=1)
[u'b']
>>> queue.clear() # remove all queue
1
>>> os.remove(filename)
```

clear(*keys=None*) [source]

Remove keys from queue. If keys is None remove all.

Returns the number of keys removed

counter = *<method-wrapper 'next' of itertools.count object at 0x377c128>*

pull(*limit=1000*) [source]

Get queued keys up to limit

push(*key_map*) [source]

Add these keys to the queue Will not insert if key already exists.

key_map:
a list of (key, priority) tuples

size = *None*

webkit Module

Interface to qt webkit for parsing JavaScript dependent webpages

class webscraping.webkit.**NetworkAccessManager**(*proxy*, *forbidden_extensions*, *allowed_regex*, *cache_size=100*, *cache_dir='.webkit_cache'*) [source]

Bases: **PyQt4.QtNetwork.QNetworkAccessManager**

Subclass QNetworkAccessManager for finer control network operations

catch_error(*eid*) [source]

Interpret the HTTP error ID received

createRequest(*operation*, *request*, *data*) [source]

`is_forbidden`(*request*) [source]

Returns whether this request is permitted by checking URL extension and regex XXX head request for mime?

`setProxy`(*proxy*) [source]

Allow setting string as proxy

class `webscraping.webkit.`**`NetworkReply`**(*parent*, *reply*) [source]

Bases: **`PyQt4.QtNetwork.QNetworkReply`**

Override QNetworkReply so can save the original data

`abort`() [source]

`applyMetaData`() [source]

`bytesAvailable`() [source]

How many bytes in the buffer are available to be read

`isSequential`() [source]

`readData`(*size*) [source]

Return up to size bytes from buffer

`readInternal`() [source]

New data available to read

class `webscraping.webkit.`**`WebPage`**(*user_agent*, *confirm=True*) [source]

Bases: **`PyQt4.QtWebKit.QWebPage`**

Override QWebPage to set User-Agent and JavaScript messages

user_agent:
the User Agent to submit

confirm:
default response to confirm dialog boxes

`javaScriptAlert`(*frame*, *message*) [source]

Override default JavaScript alert popup and print results

`javaScriptConfirm`(*frame*, *message*) [source]

Override default JavaScript confirm popup and print results

`javaScriptConsoleMessage`(*message*, *line_number*, *source_id*) [source]

Print JavaScript console messages

javaScriptPrompt(*frame*, *message*, *default*) [source]

Override default JavaScript prompt popup and print results

shouldInterruptJavaScript() [source]

Disable javascript interruption dialog box

userAgentForUrl(*url*) [source]

class `webscraping.webkit.`**WebkitBrowser**(*base_url=None*, *gui=False*, *user_agent=None*, *proxy=None*, *load_images=False*, *forbidden_extensions=None*, *allowed_regex='.*?'*, *timeout=20*, *delay=5*, *enable_plugins=False*) [source]

Bases: `PyQt4.QtWebKit.QWebView`

Render webpages using webkit

base_url:
: the domain that will be crawled

gui:
: whether to show webkit window or run headless

user_agent:
: the user-agent when downloading content

proxy:
: a QNetworkProxy to download through

load_images:
: whether to download images

forbidden_extensions
: a list of extensions to prevent downloading

allowed_regex:
: a regular expressions of URLs to allow

timeout:
: the maximum amount of seconds to wait for a request

delay:
: the minimum amount of seconds to wait between requests

attr(*pattern*, *name*, *value=None*) [source]

Set attribute if value is defined, else get

click(*pattern='input'*) [source]

Click all elements that match the pattern.

Uses standard CSS pattern matching: http://www.w3.org/TR/CSS2/selector.html

closeEvent(*event*) [source]

Catch the close window event and stop the script

current_html() [source]

Return current rendered HTML

current_url() [source]

Return current URL

data(*url*) [source]

Get data for this downloaded resource, if exists

fill(*pattern*, *value*) [source]

Set text of these elements to value

find(*pattern*) [source]

Returns whether element matching css pattern exists Note this uses CSS syntax, not Xpath

finished(*reply*) [source]

Override this method in subclasses to process downloaded urls

get(*url=None*, *html=None*, *script=None*, *num_retries=1*, *jquery=False*) [source]

Load given url in webkit and return html when loaded

url:
: the URL to load

html:
: optional HTML to set instead of downloading

script:
: some javasript to exexute that will change the loaded page (eg form submission)

num_retries:
: how many times to try downloading this URL or executing this script

jquery:
: whether to inject JQuery library into the document

inject_jquery() [source]

Inject jquery library into this webpage for easier manipulation

js(*script*) [source]

Shortcut to execute javascript on current document and return result

jsget(*script*, *num_retries=1*, *jquery=True*) [source]

Execute JavaScript that will cause page submission, and wait for page to load

run() [source]

Run the Qt event loop so can interact with the browser

screenshot(*output_file*) [source]

Take screenshot of current webpage and save results

set_proxy(*proxy*) [source]

wait(*timeout=1*, *pattern=None*) [source]

Wait for delay time

wait_load(*pattern*, *timeout=60*) [source]

Wait for this content to be loaded up to maximum timeout

webscraping.webkit.**sslErrorHandler**(*reply*, *errors*) [source]

xpath Module

This module implements a subset of the XPath standard: - tags - indices - attributes - descendants

This was created because I needed a pure Python XPath parser.

Generally XPath solutions will normalize the HTML into XHTML before selecting nodes. However this module tries to navigate the HTML structure directly without normalizing by searching for the next closing tag.

class webscraping.xpath.**Doc**(*html*, *remove=None*) [source]

Wrapper around a parsed webpage

html:

The content of webpage to parse

remove:

A list of tags to remove

```
>>> doc = Doc('<div>abc<a class="link">LINK 1</a><div><a>LINK 2</a>def</div>abc</div>gh
>>> doc.search('/div/a')
['LINK 1', 'LINK 3']
>>> doc.search('/div/a[@class="link"]')
```

```
['LINK 1']
>>> doc.search('/div[1]//a')
['LINK 1', 'LINK 2']
>>> doc.search('/div/a/@class')
['link', '']
>>> doc.search('/div[-1]/a')
['LINK 3']
```

```
>>> # test searching unicode
>>> doc = Doc(u'<a href="http://www.google.com" class="flink">google</a>')
>>> doc.get('//a[@class="flink"]')
u'google'
```

```
>>> # test finding just the first instance for a large amount of content
>>> doc = Doc('<div><span>content</span></div>' * 10000)
>>> doc.get('//span')
'content'
```

```
>>> # test extracting attribute of self closing tag
>>> Doc('<div><img src="img.png"></div>').get('/div/img/@src')
'img.png'
```

```
>>> # test extracting attribute after self closing tag
>>> Doc('<div><br><p>content</p></div>').get('/div/p')
'content'
```

get(*xpath*) [source]

Return the first result from this XPath selection

parse(*xpath*) [source]

Parse the xpath into: counter, separator, tag, index, and attributes

```
>>> doc = Doc('')
>>> doc.parse('/div[1]//span[@class="text"]')
[(0, '', 'div', 1, []), (1, '/', 'span', None, [('class', 'text')])]
>>> doc.parse('//li[-2]')
[(0, '/', 'li', -2, [])]
>>> doc.parse('//option[@selected]')
[(0, '/', 'option', None, [('selected', None)])]
>>> doc.parse('/div[@id="content"]//span[1][@class="text"][@title=""]/a')
[(0, '', 'div', None, [('id', 'content')]), (1, '/', 'span', 1, [('class', 'text')
```

search(*xpath*) [source]

Return all results from this XPath selection

class `webscraping.xpath.`**Form**(*form*) [source]

Helper class for filling and submitting forms

submit(*D*, *action*, ***argv*) [source] *class* `webscraping.xpath.`**`Tree`**(*html*, ***kwargs*) [source]

`get`(*path*) [source]

`search`(*path*) [source]

`tostring`(*node*) [source]

`webscraping.xpath.`**`find_children`**(*html*, *tag*, *remove=None*) [source]

Find children with this tag type

`webscraping.xpath.`**`get`**(*html*, *xpath*, *remove=('br', 'hr')*) [source]

Return first element from XPath search of HTML

`webscraping.xpath.`**`get_links`**(*html*, *url=None*, *local=True*, *external=True*) [source]

Return all links from html and convert relative to absolute if source url is provided

html:
: HTML to parse

url:
: optional URL for determining path of relative links

local:
: whether to include links from same domain

external:
: whether to include linkes from other domains

`webscraping.xpath.`**`search`**(*html*, *xpath*, *remove=('br', 'hr')*) [source]

Return all elements from XPath search of HTML