SEP	14
Title	CrawlSpider v2
Author	Insophia Team
Created	2010-01-22
Updated	2010-02-04
Status	Final. Partially implemented but discarded because of lack of use in r2632

SEP-014 - CrawlSpider v2

This SEP proposes a rewrite of Scrapy `CrawlSpider` and related components

Current flaws and inconsistencies

1. Request's callbacks are hard to persist.
2. Link extractors are inflexible and hard to maintain, link processing/filtering is tightly coupled. (e.g. canonicalize)
3. Isn't possible to crawl an url directly from command line because the Spider does not know which callback use.

These flaws will be corrected by the changes proposed in this SEP.

Proposed API Changes

- Separate the functionality of Rule-LinkExtractor-Callback
- Separate the functionality of LinkExtractor to Request Extractor and Request Processor
- Separate the process of determining response callback and the extraction of new requests (link extractors)
- The callback will be determine by Matcher Objects on request/response objects

Matcher Objects

Matcher Objects (aka Matcher) are responsible for determining if given request or response matches an arbitrary criteria. The Matcher receives as argument the request or the response, giving a powerful access to all request/response attributes.

In the current `CrawlSpider`, the Rule Object has the responsibility to determine the callback of given extractor, and the link extractor contains the url pattern (aka regex). Now the Matcher will contain only the pattern or criteria to determine which request/response will execute any action. See below Spider Rules.

Request Extractors

Request Extractors takes response object and determines which requests follow.

This is an enhancement to `LinkExtractors` which returns urls (links), Request Extractors return Request objects.

Request Processors

Request Processors takes requests objects and can perform any action to them, like filtering or modifying on the fly.

The current `LinkExtractor` had integrated link processing, like canonicalize. Request Processors can be reutilized and applied in series.

Request Generator

Request Generator is the decoupling of the `CrawlSpider`'s method _request_to_follow(). Request Generator takes the response object and applies the Request Extractors and Request Processors.

Rules Manager

The Rules are a definition of Rule objects containing Matcher Objects and callback.

The Legacy Rules were used to perform the link extraction and attach the callback to the generated Request object. The proposed new Rules will be used to determine the callback for given response. This opens a whole of opportunities, like determine the callback for given url, and persist the queue of Request objects because the callback is determined the matching the Response object against the Rules.

Usage Examples

Basic Crawling

```
#!python
#
# Basic Crawling
#
class SampleSpider(CrawlSpider):
    rules = [
```

```
        # The dispatcher uses first-match policy
        Rule(UrlRegexMatch(r'product\.html\?id=\d+'), 'parse_item', follow=False),
        # by default, if the first param is string is wrapped into UrlRegexMatch
        Rule(r'.+', 'parse_page'),
        ]

    request_extractors = [
        # crawl all links looking for products and images
        SgmlRequestExtractor(),
        ]

    request_processors = [
        # canonicalize all requests' urls
        Canonicalize(),
        ]

    def parse_item(self, response):
        # parse and extract items from response
        pass

    def parse_page(self, response):
        # extract images on all pages
        pass
```

Custom Processor and External Callback

```
#!python
#
# Using external callbacks
#

# Custom Processor
def filter_today_links(requests):
    # only crawl today links
    today = datetime.datetime.today().strftime('%Y-%m-%d')
    return [r for r in requests if today in r.url]

# Callback defined out of spider
def my_external_callback(response):
    # process item
    pass

class SampleSpider(CrawlSpider):
    rules = [
        # The dispatcher uses first-match policy
        Rule(UrlRegexMatch(r'/news/(.+)/'), my_external_callback),
        ]

    request_extractors = [
        RegexRequestExtractor(r'/sections/.+'),
        RegexRequestExtractor(r'/news/.+'),
        ]

    request_processors = [
        # canonicalize all requests' urls
        Canonicalize(),
        filter_today_links,
        ]
```

Implementation

Work-in-progress

Package Structure

```
contrib_exp
    |- crawlspider/
        |- spider.py
            |- CrawlSpider
        |- rules.py
            |- Rule
            |- CompiledRule
            |- RulesManager
        |- reqgen.py
            |- RequestGenerator
        |- reqproc.py
            |- Canonicalize
            |- Unique
            |- ...
        |- reqext.py
```

```
            |- SgmlRequestExtractor
            |- RegexRequestExtractor
            |- ...
        |- matchers.py
            |- BaseMatcher
            |- UrlMatcher
            |- UrlRegexMatcher
            |- ...
```

Request/Response Matchers

```
#!python
"""
Request/Response Matchers

Perform evaluation to Request or Response attributes
"""

class BaseMatcher(object):
    """Base matcher. Returns True by default."""

    def matches_request(self, request):
        """Performs Request Matching"""
        return True

    def matches_response(self, response):
        """Performs Response Matching"""
        return True


class UrlMatcher(BaseMatcher):
    """Matches URL attribute"""

    def __init__(self, url):
        """Initialize url attribute"""
        self._url = url

    def matches_url(self, url):
        """Returns True if given url is equal to matcher's url"""
        return self._url url

    def matches_request(self, request):
        """Returns True if Request's url matches initial url"""
        return self.matches_url(request.url)

    def matches_response(self, response):
        """REturns True if Response's url matches initial url"""
        return self.matches_url(response.url)


class UrlRegexMatcher(UrlMatcher):
    """Matches URL using regular expression"""

    def __init__(self, regex, flags=0):
        """Initialize regular expression"""
        self._regex = re.compile(regex, flags)

    def matches_url(self, url):
        """Returns True if url matches regular expression"""
        return self._regex.search(url) is not None
```

Request Extractor

```
#!python
#
# Requests Extractor
# Extractors receive response and return list of Requests
#

class BaseSgmlRequestExtractor(FixedSGMLParser):
    """Base SGML Request Extractor"""

    def __init__(self, tag='a', attr='href'):
        """Initialize attributes"""
        FixedSGMLParser.__init__(self)

        self.scan_tag = tag if callable(tag) else lambda t: t tag
        self.scan_attr = attr if callable(attr) else lambda a: a attr
        self.current_request = None

    def extract_requests(self, response):
```

```
        """Returns list of requests extracted from response"""
        return self._extract_requests(response.body, response.url,
                                      response.encoding)

    def _extract_requests(self, response_text, response_url, response_encoding):
        """Extract requests with absolute urls"""
        self.reset()
        self.feed(response_text)
        self.close()

        base_url = self.base_url if self.base_url else response_url
        self._make_absolute_urls(base_url, response_encoding)
        self._fix_link_text_encoding(response_encoding)

        return self.requests

    def _make_absolute_urls(self, base_url, encoding):
        """Makes all request's urls absolute"""
        for req in self.requests:
            url = req.url
            # make absolute url
            url = urljoin_rfc(base_url, url, encoding)
            url = safe_url_string(url, encoding)
            # replace in-place request's url
            req.url = url

    def _fix_link_text_encoding(self, encoding):
        """Convert link_text to unicode for each request"""
        for req in self.requests:
            req.meta.setdefault('link_text', '')
            req.meta['link_text'] = str_to_unicode(req.meta['link_text'],
                                                   encoding)

    def reset(self):
        """Reset state"""
        FixedSGMLParser.reset(self)
        self.requests = []
        self.base_url = None

    def unknown_starttag(self, tag, attrs):
        """Process unknown start tag"""
        if 'base' tag:
            self.base_url = dict(attrs).get('href')

        if self.scan_tag(tag):
            for attr, value in attrs:
                if self.scan_attr(attr):
                    if value is not None:
                        req = Request(url=value)
                        self.requests.append(req)
                        self.current_request = req

    def unknown_endtag(self, tag):
        """Process unknown end tag"""
        self.current_request = None

    def handle_data(self, data):
        """Process data"""
        current = self.current_request
        if current and not 'link_text' in current.meta:
            current.meta['link_text'] = data.strip()


class SgmlRequestExtractor(BaseSgmlRequestExtractor):
    """SGML Request Extractor"""

    def __init__(self, tags=None, attrs=None):
        """Initialize with custom tag & attribute function checkers"""
        # defaults
        tags = tuple(tags) if tags else ('a', 'area')
        attrs = tuple(attrs) if attrs else ('href', )

        tag_func = lambda x: x in tags
        attr_func = lambda x: x in attrs
        BaseSgmlRequestExtractor.__init__(self, tag=tag_func, attr=attr_func)


class XPathRequestExtractor(SgmlRequestExtractor):
    """SGML Request Extractor with XPath restriction"""

    def __init__(self, restrict_xpaths, tags=None, attrs=None):
```

```
        """Initialize XPath restrictions"""
        self.restrict_xpaths = tuple(arg_to_iter(restrict_xpaths))
        SgmlRequestExtractor.__init__(self, tags, attrs)

    def extract_requests(self, response):
        """Restrict to XPath regions"""
        hxs = HtmlXPathSelector(response)
        fragments = (''.join(
                            html_frag for html_frag in hxs.select(xpath).extract()
                        ) for xpath in self.restrict_xpaths)
        html_slice = ''.join(html_frag for html_frag in fragments)
        return self._extract_requests(html_slice, response.url,
                                      response.encoding)
```

Request Processor

```
#!python
#
# Request Processors
# Processors receive list of requests and return list of requests
#
"""Request Processors"""

class Canonicalize(object):
    """Canonicalize Request Processor"""

    def __call__(self, requests):
        """Canonicalize all requests' urls"""
        for req in requests:
            # replace in-place
            req.url = canonicalize_url(req.url)
            yield req


class Unique(object):
    """Filter duplicate Requests"""

    def __init__(self, *attributes):
        """Initialize comparison attributes"""
        self._attributes = attributes or ['url']

    def _requests_equal(self, req1, req2):
        """Attribute comparison helper"""
        for attr in self._attributes:
            if getattr(req1, attr) != getattr(req2, attr):
                return False
        # all attributes equal
        return True

    def _request_in(self, request, requests_seen):
        """Check if request is in given requests seen list"""
        for seen in requests_seen:
            if self._requests_equal(request, seen):
                return True
        # request not seen
        return False

    def __call__(self, requests):
        """Filter seen requests"""
        # per-call duplicates filter
        requests_seen = set()
        for req in requests:
            if not self._request_in(req, requests_seen):
                yield req
                # registry seen request
                requests_seen.add(req)


class FilterDomain(object):
    """Filter request's domain"""

    def __init__(self, allow=(), deny=()):
         """Initialize allow/deny attributes"""
         self.allow = tuple(arg_to_iter(allow))
         self.deny = tuple(arg_to_iter(deny))

    def __call__(self, requests):
        """Filter domains"""
        processed = (req for req in requests)

        if self.allow:
```

```
            processed = (req for req in requests
                         if url_is_from_any_domain(req.url, self.allow))
        if self.deny:
            processed = (req for req in requests
                         if not url_is_from_any_domain(req.url, self.deny))

        return processed


class FilterUrl(object):
    """Filter request's url"""

    def __init__(self, allow=(), deny=()):
        """Initialize allow/deny attributes"""
        _re_type = type(re.compile('', 0))

        self.allow_res = [x if isinstance(x, _re_type) else re.compile(x)
                          for x in arg_to_iter(allow)]
        self.deny_res = [x if isinstance(x, _re_type) else re.compile(x)
                         for x in arg_to_iter(deny)]

    def __call__(self, requests):
        """Filter request's url based on allow/deny rules"""
        #TODO: filter valid urls here?
        processed = (req for req in requests)

        if self.allow_res:
            processed = (req for req in requests
                         if self._matches(req.url, self.allow_res))
        if self.deny_res:
            processed = (req for req in requests
                         if not self._matches(req.url, self.deny_res))

        return processed

    def _matches(self, url, regexs):
        """Returns True if url matches any regex in given list"""
        return any(r.search(url) for r in regexs)
```

Rule Object

```
#!python
#
# Dispatch Rules classes
# Manage Rules (Matchers + Callbacks)
#
class Rule(object):
    """Crawler Rule"""
    def __init__(self, matcher, callback=None, cb_args=None,
                 cb_kwargs=None, follow=True):
        """Store attributes"""
        self.matcher = matcher
        self.callback = callback
        self.cb_args = cb_args if cb_args else ()
        self.cb_kwargs = cb_kwargs if cb_kwargs else {}
        self.follow = follow

#
# Rules Manager takes list of Rule objects and normalize matcher and callback
# into CompiledRule
#
class CompiledRule(object):
    """Compiled version of Rule"""
    def __init__(self, matcher, callback=None, follow=False):
        """Initialize attributes checking type"""
        assert isinstance(matcher, BaseMatcher)
        assert callback is None or callable(callback)
        assert isinstance(follow, bool)

        self.matcher = matcher
        self.callback = callback
        self.follow = follow
```

Rules Manager

```
#!python
#
# Handles rules matcher/callbacks
# Resolve rule for given response
#
class RulesManager(object):
```

```
    """Rules Manager"""
    def __init__(self, rules, spider, default_matcher=UrlRegexMatcher):
        """Initialize rules using spider and default matcher"""
        self._rules = tuple()

        # compile absolute/relative-to-spider callbacks"""
        for rule in rules:
            # prepare matcher
            if isinstance(rule.matcher, BaseMatcher):
                matcher = rule.matcher
            else:
                # matcher not BaseMatcher, check for string
                if isinstance(rule.matcher, basestring):
                    # instance default matcher
                    matcher = default_matcher(rule.matcher)
                else:
                    raise ValueError('Not valid matcher given %r in %r' \
                                % (rule.matcher, rule))

            # prepare callback
            if callable(rule.callback):
                callback = rule.callback
            elif not rule.callback is None:
                # callback from spider
                callback = getattr(spider, rule.callback)

                if not callable(callback):
                    raise AttributeError('Invalid callback %r can not be resolved' \
                                        % callback)
            else:
                callback = None

            if rule.cb_args or rule.cb_kwargs:
                # build partial callback
                callback = partial(callback, *rule.cb_args, **rule.cb_kwargs)

            # append compiled rule to rules list
            crule = CompiledRule(matcher, callback, follow=rule.follow)
            self._rules += (crule, )

    def get_rule(self, response):
        """Returns first rule that matches response"""
        for rule in self._rules:
            if rule.matcher.matches_response(response):
                return rule
```

Request Generator

```
#!python
#
# Request Generator
# Takes response and generate requests using extractors and processors
#
class RequestGenerator(object):
    def __init__(self, req_extractors, req_processors, callback):
        self._request_extractors = req_extractors
        self._request_processors = req_processors
        self.callback = callback

    def generate_requests(self, response):
        """
        Extract and process new requets from response
        """
        requests = []
        for ext in self._request_extractors:
            requets.extend(ext.extract_requests(response))

        for proc in self._request_processors:
            requests = proc(requests)

        for request in requests:
            yield request.replace(callback=self.callback)
```

CrawlSpider

```
#!python
#
# Spider
#
class CrawlSpider(InitSpider):
    """CrawlSpider v2"""
```

```
request_extractors = []
request_processors = []
rules = []

def __init__(self):
    """Initialize dispatcher"""
    super(CrawlSpider, self).__init__()

    # wrap rules
    self._rulesman = RulesManager(self.rules, spider=self)
    # generates new requests with given callback
    self._reqgen = RequestGenerator(self.request_extractors,
                                    self.request_processors,
                                    self.parse)

def parse(self, response):
    """Dispatch callback and generate requests"""
    # get rule for response
    rule = self._rulesman.get_rule(response)
    if rule:
        # dispatch callback if set
        if rule.callback:
            output = iterate_spider_output(rule.callback(response))
            for req_or_item in output:
                yield req_or_item

        if rule.follow:
            for req in self._reqgen.generate_requests(response):
                yield req
```