Spiders

Spiders are classes which define how a certain site (or a group of sites) will be scraped, including how to perform the crawl (i.e. follow links) and how to extract structured data from their pages (i.e. scraping items). In other words, Spiders are the place where you define the custom behaviour for crawling and parsing pages for a particular site (or, in some cases, a group of sites).

For spiders, the scraping cycle goes through something like this:

1. You start by generating the initial Requests to crawl the first URLs, and specify a callback function to be called with the response downloaded from those requests.

   The first requests to perform are obtained by calling the `start_requests()` method which (by default) generates `Request` for the URLs specified in the `start_urls` and the `parse` method as callback function for the Requests.
2. In the callback function, you parse the response (web page) and return either dicts with extracted data, `Item` objects, `Request` objects, or an iterable of these objects. Those Requests will also contain a callback (maybe the same) and will then be downloaded by Scrapy and then their response handled by the specified callback.
3. In callback functions, you parse the page contents, typically using Selectors (but you can also use BeautifulSoup, lxml or whatever mechanism you prefer) and generate items with the parsed data.
4. Finally, the items returned from the spider will be typically persisted to a database (in some Item Pipeline) or written to a file using Feed exports.

Even though this cycle applies (more or less) to any kind of spider, there are different kinds of default spiders bundled into Scrapy for different purposes. We will talk about those types here.

scrapy.Spider

class scrapy.spiders.Spider

This is the simplest spider, and the one from which every other spider must inherit (including spiders that come bundled with Scrapy, as well as spiders that you write yourself). It doesn't provide any special functionality. It just provides a default `start_requests()` implementation which sends requests from the `start_urls` spider attribute and calls the spider's method `parse` for each of the resulting responses.

name

A string which defines the name for this spider. The spider name is how the spider is located (and instantiated) by Scrapy, so it must be unique. However, nothing prevents you from instantiating more than one instance of the same spider. This is the most important spider attribute and it's required.

If the spider scrapes a single domain, a common practice is to name the spider after the domain, with or without the TLD. So, for example, a spider that crawls `mywebsite.com` would often be called `mywebsite`.

Note

In Python 2 this must be ASCII only.

allowed_domains

An optional list of strings containing domains that this spider is allowed to crawl. Requests for URLs not belonging to the domain names specified in this list (or their subdomains) won't be followed if `OffsiteMiddleware` is enabled.

Let's say your target url is `https://www.example.com/1.html`, then add `'example.com'` to the list.

start_urls

A list of URLs where the spider will begin to crawl from, when no particular URLs are specified. So, the first pages downloaded will be those listed here. The subsequent URLs will be generated successively from data contained in the start URLs.

custom_settings

A dictionary of settings that will be overridden from the project wide configuration when running this spider. It must be defined as a class attribute since the settings are updated before instantiation.

For a list of available built-in settings see: Built-in settings reference.

crawler

This attribute is set by the `from_crawler()` class method after initializating the class, and links to the `Crawler` object to which this spider instance is bound.

Crawlers encapsulate a lot of components in the project for their single entry access (such as extensions, middlewares, signals managers, etc). See Crawler API to know more about them.

`settings`

Configuration for running this spider. This is a `Settings` instance, see the Settings topic for a detailed introduction on this subject.

`logger`

Python logger created with the Spider's `name`. You can use it to send log messages through it as described on Logging from Spiders.

`from_crawler`(crawler, *args, **kwargs)

This is the class method used by Scrapy to create your spiders.

You probably won't need to override this directly because the default implementation acts as a proxy to the `__init__()` method, calling it with the given arguments args and named arguments kwargs.

Nonetheless, this method sets the `crawler` and `settings` attributes in the new instance so they can be accessed later inside the spider's code.

Parameters:
- **crawler** (`Crawler` instance) – crawler to which the spider will be bound
- **args** (list) – arguments passed to the `__init__()` method
- **kwargs** (dict) – keyword arguments passed to the `__init__()` method

`start_requests`()

This method must return an iterable with the first Requests to crawl for this spider. It is called by Scrapy when the spider is opened for scraping. Scrapy calls it only once, so it is safe to implement `start_requests()` as a generator.

The default implementation generates `Request(url, dont_filter=True)` for each url in `start_urls`.

If you want to change the Requests used to start scraping a domain, this is the method to override. For example, if you need to start by logging in using a POST request, you could do:

```
class MySpider(scrapy.Spider):
    name = 'myspider'

    def start_requests(self):
        return [scrapy.FormRequest("http://www.example.com/login",
                                   formdata={'user': 'john', 'pass': 'secret'},
```

```
                                    callback=self.logged_in)]

    def logged_in(self, response):
        # here you would extract links to follow and return Requests for
        # each of them, with another callback
        pass
```

parse(response)

This is the default callback used by Scrapy to process downloaded responses, when their requests don't specify a callback.

The `parse` method is in charge of processing the response and returning scraped data and/or more URLs to follow. Other Requests callbacks have the same requirements as the `Spider` class.

This method, as well as any other Request callback, must return an iterable of `Request` and/or dicts or `Item` objects.

Parameters: **response** (`Response`) – the response to parse

log(message [, level, component])

Wrapper that sends a log message through the Spider's `logger`, kept for backwards compatibility. For more information see Logging from Spiders.

closed(reason)

Called when the spider closes. This method provides a shortcut to signals.connect() for the `spider_closed` signal.

Let's see an example:

```
import scrapy


class MySpider(scrapy.Spider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = [
        'http://www.example.com/1.html',
        'http://www.example.com/2.html',
        'http://www.example.com/3.html',
    ]

    def parse(self, response):
        self.logger.info('A response from %s just arrived!', response.url)
```

Return multiple Requests and items from a single callback:

```
import scrapy
```

```
class MySpider(scrapy.Spider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = [
        'http://www.example.com/1.html',
        'http://www.example.com/2.html',
        'http://www.example.com/3.html',
    ]

    def parse(self, response):
        for h3 in response.xpath('//h3').extract():
            yield {"title": h3}

        for url in response.xpath('//a/@href').extract():
            yield scrapy.Request(url, callback=self.parse)
```

Instead of `start_urls` you can use `start_requests()` directly; to give data more structure you can use Items:

```
import scrapy
from myproject.items import MyItem

class MySpider(scrapy.Spider):
    name = 'example.com'
    allowed_domains = ['example.com']

    def start_requests(self):
        yield scrapy.Request('http://www.example.com/1.html', self.parse)
        yield scrapy.Request('http://www.example.com/2.html', self.parse)
        yield scrapy.Request('http://www.example.com/3.html', self.parse)

    def parse(self, response):
        for h3 in response.xpath('//h3').extract():
            yield MyItem(title=h3)

        for url in response.xpath('//a/@href').extract():
            yield scrapy.Request(url, callback=self.parse)
```

Spider arguments

Spiders can receive arguments that modify their behaviour. Some common uses for spider arguments are to define the start URLs or to restrict the crawl to certain sections of the site, but they can be used to configure any functionality of the spider.

Spider arguments are passed through the `crawl` command using the `-a` option. For example:

```
scrapy crawl myspider -a category=electronics
```

Spiders can access arguments in their __init__ methods:

```
import scrapy

class MySpider(scrapy.Spider):
    name = 'myspider'

    def __init__(self, category=None, *args, **kwargs):
        super(MySpider, self).__init__(*args, **kwargs)
        self.start_urls = ['http://www.example.com/categories/%s' % category]
        # ...
```

The default __init__ method will take any spider arguments and copy them to the spider as attributes. The above example can also be written as follows:

```
import scrapy

class MySpider(scrapy.Spider):
    name = 'myspider'

    def start_requests(self):
        yield scrapy.Request('http://www.example.com/categories/%s' % self.category)
```

Keep in mind that spider arguments are only strings. The spider will not do any parsing on its own. If you were to set the start_urls attribute from the command line, you would have to parse it on your own into a list using something like ast.literal_eval or json.loads and then set it as an attribute. Otherwise, you would cause iteration over a start_urls string (a very common python pitfall) resulting in each character being seen as a separate url.

A valid use case is to set the http auth credentials used by `HttpAuthMiddleware` or the user agent used by `UserAgentMiddleware`:

```
scrapy crawl myspider -a http_user=myuser -a http_pass=mypassword -a user_agent=mybot
```

Spider arguments can also be passed through the Scrapyd `schedule.json` API. See Scrapyd documentation.

Generic Spiders

Scrapy comes with some useful generic spiders that you can use to subclass your spiders from. Their aim is to provide convenient functionality for a few common scraping cases, like following all links on a site based on certain rules, crawling from Sitemaps, or parsing an XML/CSV feed.

For the examples used in the following spiders, we'll assume you have a project with a `TestItem` declared in a `myproject.items` module:

```
import scrapy

class TestItem(scrapy.Item):
    id = scrapy.Field()
    name = scrapy.Field()
    description = scrapy.Field()
```

CrawlSpider

class **scrapy.spiders.CrawlSpider**

This is the most commonly used spider for crawling regular websites, as it provides a convenient mechanism for following links by defining a set of rules. It may not be the best suited for your particular web sites or project, but it's generic enough for several cases, so you can start from it and override it as needed for more custom functionality, or just implement your own spider.

Apart from the attributes inherited from Spider (that you must specify), this class supports a new attribute:

rules

Which is a list of one (or more) `Rule` objects. Each `Rule` defines a certain behaviour for crawling the site. Rules objects are described below. If multiple rules match the same link, the first one will be used, according to the order they're defined in this attribute.

This spider also exposes an overrideable method:

parse_start_url(response)

This method is called for the start_urls responses. It allows to parse the initial responses and must return either an `Item` object, a `Request` object, or an iterable containing any of them.

Crawling rules

class **scrapy.spiders.Rule**(link_extractor, callback=None, cb_kwargs=None, follow=None, process_links=None, process_request=None)

`link_extractor` is a Link Extractor object which defines how links will be extracted from each crawled page.

`callback` is a callable or a string (in which case a method from the spider object with that name will be used) to be called for each link extracted with the specified link_extractor. This callback receives a response as its first argument and must return a list containing `Item` and/or `Request` objects (or any subclass of them).

Warning

When writing crawl spider rules, avoid using `parse` as callback, since the `CrawlSpider` uses the `parse` method itself to implement its logic. So if you override the `parse` method, the crawl

spider will no longer work.

`cb_kwargs` is a dict containing the keyword arguments to be passed to the callback function.

`follow` is a boolean which specifies if links should be followed from each response extracted with this rule. If `callback` is None `follow` defaults to `True`, otherwise it defaults to `False`.

`process_links` is a callable, or a string (in which case a method from the spider object with that name will be used) which will be called for each list of links extracted from each response using the specified `link_extractor`. This is mainly used for filtering purposes.

`process_request` is a callable, or a string (in which case a method from the spider object with that name will be used) which will be called with every request extracted by this rule, and must return a request or None (to filter out the request).

CrawlSpider example

Let's now take a look at an example CrawlSpider with rules:

```
import scrapy
from scrapy.spiders import CrawlSpider, Rule
from scrapy.linkextractors import LinkExtractor

class MySpider(CrawlSpider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = ['http://www.example.com']

    rules = (
        # Extract links matching 'category.php' (but not matching 'subsection.php')
        # and follow links from them (since no callback means follow=True by default).
        Rule(LinkExtractor(allow=('category\.php', ), deny=('subsection\.php', ))),

        # Extract links matching 'item.php' and parse them with the spider's method parse_item
        Rule(LinkExtractor(allow=('item\.php', )), callback='parse_item'),
    )

    def parse_item(self, response):
        self.logger.info('Hi, this is an item page! %s', response.url)
        item = scrapy.Item()
        item['id'] = response.xpath('//td[@id="item_id"]/text()').re(r'ID: (\d+)')
        item['name'] = response.xpath('//td[@id="item_name"]/text()').extract()
        item['description'] = response.xpath('//td[@id="item_description"]/text()').extract()
        return item
```

This spider would start crawling example.com's home page, collecting category links, and item links, parsing the latter with the `parse_item` method. For each item response, some data will be extracted from the HTML using XPath, and an `Item` will be filled with it.

XMLFeedSpider

class scrapy.spiders.XMLFeedSpider

XMLFeedSpider is designed for parsing XML feeds by iterating through them by a certain node name. The iterator can be chosen from: `iternodes`, `xml`, and `html`. It's recommended to use the `iternodes` iterator for performance reasons, since the `xml` and `html` iterators generate the whole DOM at once in order to parse it. However, using `html` as the iterator may be useful when parsing XML with bad markup.

To set the iterator and the tag name, you must define the following class attributes:

iterator

A string which defines the iterator to use. It can be either:

- `'iternodes'` - a fast iterator based on regular expressions
- `'html'` - an iterator which uses `Selector`. Keep in mind this uses DOM parsing and must load all DOM in memory which could be a problem for big feeds
- `'xml'` - an iterator which uses `Selector`. Keep in mind this uses DOM parsing and must load all DOM in memory which could be a problem for big feeds

It defaults to: `'iternodes'`.

itertag

A string with the name of the node (or element) to iterate in. Example:

```
itertag = 'product'
```

namespaces

A list of `(prefix, uri)` tuples which define the namespaces available in that document that will be processed with this spider. The `prefix` and `uri` will be used to automatically register namespaces using the `register_namespace()` method.

You can then specify nodes with namespaces in the `itertag` attribute.

Example:

```
class YourSpider(XMLFeedSpider):

    namespaces = [('n', 'http://www.sitemaps.org/schemas/sitemap/0.9')]
    itertag = 'n:url'
    # ...
```

Apart from these new attributes, this spider has the following overrideable methods too:

adapt_response(response)

A method that receives the response as soon as it arrives from the spider middleware, before the spider starts parsing it. It can be used to modify the response body before parsing it. This method receives a response and also returns a response (it could be the same or another one).

parse_node(response, selector)

This method is called for the nodes matching the provided tag name (`itertag`). Receives the response and an `Selector` for each node. Overriding this method is mandatory. Otherwise, you spider won't work. This method must return either a `Item` object, a `Request` object, or an iterable containing any of them.

process_results(response, results)

This method is called for each result (item or request) returned by the spider, and it's intended to perform any last time processing required before returning the results to the framework core, for example setting the item IDs. It receives a list of results and the response which originated those results. It must return a list of results (Items or Requests).

XMLFeedSpider example

These spiders are pretty easy to use, let's have a look at one example:

```
from scrapy.spiders import XMLFeedSpider
from myproject.items import TestItem

class MySpider(XMLFeedSpider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = ['http://www.example.com/feed.xml']
    iterator = 'iternodes'  # This is actually unnecessary, since it's the default value
    itertag = 'item'

    def parse_node(self, response, node):
        self.logger.info('Hi, this is a <%s> node!: %s', self.itertag, ''.join(node.extract()))

        item = TestItem()
        item['id'] = node.xpath('@id').extract()
        item['name'] = node.xpath('name').extract()
        item['description'] = node.xpath('description').extract()
        return item
```

Basically what we did up there was to create a spider that downloads a feed from the given `start_urls`, and then iterates through each of its `item` tags, prints them out, and stores some random data in an `Item`.

CSVFeedSpider

class **scrapy.spiders.CSVFeedSpider**

This spider is very similar to the XMLFeedSpider, except that it iterates over rows, instead of nodes. The method that gets called in each iteration is `parse_row()`.

`delimiter`

A string with the separator character for each field in the CSV file Defaults to `','` (comma).

`quotechar`

A string with the enclosure character for each field in the CSV file Defaults to `'"'` (quotation mark).

`headers`

A list of the column names in the CSV file.

`parse_row`(response, row)

Receives a response and a dict (representing each row) with a key for each provided (or detected) header of the CSV file. This spider also gives the opportunity to override `adapt_response` and `process_results` methods for pre- and post-processing purposes.

CSVFeedSpider example

Let's see an example similar to the previous one, but using a `CSVFeedSpider`:

```
from scrapy.spiders import CSVFeedSpider
from myproject.items import TestItem

class MySpider(CSVFeedSpider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = ['http://www.example.com/feed.csv']
    delimiter = ';'
    quotechar = "'"
    headers = ['id', 'name', 'description']

    def parse_row(self, response, row):
        self.logger.info('Hi, this is a row!: %r', row)

        item = TestItem()
        item['id'] = row['id']
        item['name'] = row['name']
        item['description'] = row['description']
        return item
```

SitemapSpider

class **scrapy.spiders.SitemapSpider**

SitemapSpider allows you to crawl a site by discovering the URLs using Sitemaps.

It supports nested sitemaps and discovering sitemap urls from robots.txt.

`sitemap_urls`

A list of urls pointing to the sitemaps whose urls you want to crawl.

You can also point to a robots.txt and it will be parsed to extract sitemap urls from it.

`sitemap_rules`

A list of tuples `(regex, callback)` where:

- `regex` is a regular expression to match urls extracted from sitemaps. `regex` can be either a str or a compiled regex object.
- callback is the callback to use for processing the urls that match the regular expression. `callback` can be a string (indicating the name of a spider method) or a callable.

For example:

```
sitemap_rules = [('/product/', 'parse_product')]
```

Rules are applied in order, and only the first one that matches will be used.

If you omit this attribute, all urls found in sitemaps will be processed with the `parse` callback.

`sitemap_follow`

A list of regexes of sitemap that should be followed. This is is only for sites that use Sitemap index files that point to other sitemap files.

By default, all sitemaps are followed.

`sitemap_alternate_links`

Specifies if alternate links for one `url` should be followed. These are links for the same website in another language passed within the same `url` block.

For example:

```
<url>
    <loc>http://example.com/</loc>
    <xhtml:link rel="alternate" hreflang="de" href="http://example.com/de"/>
</url>
```

With `sitemap_alternate_links` set, this would retrieve both URLs. With `sitemap_alternate_links` disabled, only `http://example.com/` would be retrieved.

Default is `sitemap_alternate_links` disabled.

SitemapSpider examples

Simplest example: process all urls discovered through sitemaps using the `parse` callback:

```
from scrapy.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/sitemap.xml']

    def parse(self, response):
        pass # ... scrape item here ...
```

Process some urls with certain callback and other urls with a different callback:

```
from scrapy.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/sitemap.xml']
    sitemap_rules = [
        ('/product/', 'parse_product'),
        ('/category/', 'parse_category'),
    ]

    def parse_product(self, response):
        pass # ... scrape product ...

    def parse_category(self, response):
        pass # ... scrape category ...
```

Follow sitemaps defined in the robots.txt file and only follow sitemaps whose url contains `/sitemap_shop`:

```
from scrapy.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
        ('/shop/', 'parse_shop'),
    ]
    sitemap_follow = ['/sitemap_shops']

    def parse_shop(self, response):
        pass # ... scrape shop here ...
```

Combine SitemapSpider with other sources of urls:

```
from scrapy.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
```

```
        ('/shop/', 'parse_shop'),
    ]

    other_urls = ['http://www.example.com/about']

    def start_requests(self):
        requests = list(super(MySpider, self).start_requests())
        requests += [scrapy.Request(x, self.parse_other) for x in self.other_urls]
        return requests

    def parse_shop(self, response):
        pass # ... scrape shop here ...

    def parse_other(self, response):
        pass # ... scrape other here ...
```

« Previous

Next »

© Copyright 2008-2016, Scrapy developers. Revision aa83e159.

Built with Sphinx using a theme provided by Read the Docs.