Scrapy is a powerful and popular open-source web crawling framework written in Python. It is used to extract data from websites and process it as per your requirements. Here’s a brief overview to get you started:

**Key Features:**

* **Fast and Efficient**: Scrapy is designed to be fast and efficient, making it ideal for large-scale web scraping projects.
* **Extensible**: It is highly customizable, allowing you to extend its functionality with middlewares, pipelines, and custom settings.
* **Built-in Data Export**: You can export scraped data in various formats such as JSON, CSV, and XML.
* **Handling Requests**: Scrapy handles requests asynchronously, which speeds up the scraping process.

**Basic Usage:**

1. **Installation**:

pip install scrapy

1. **Creating a New Project**:

scrapy startproject myproject

cd myproject

1. **Generating a Spider**:

scrapy genspider example example.com

1. **Writing a Spider**: Here’s a simple example of a spider that scrapes quotes from a website:
2. import scrapy
3. class QuotesSpider(scrapy.Spider):
4. name = "quotes"
5. start\_urls = [
6. 'http://quotes.toscrape.com/page/1/',
7. ]
8. def parse(self, response):
9. for quote in response.css('div.quote'):
10. yield {
11. 'text': quote.css('span.text::text').get(),
12. 'author': quote.css('small.author::text').get(),
13. 'tags': quote.css('div.tags a.tag::text').getall(),
14. }
15. next\_page = response.css('li.next a::attr(href)').get()
16. if next\_page is not None:
17. yield response.follow(next\_page, self.parse)
18. **Running the Spider**:

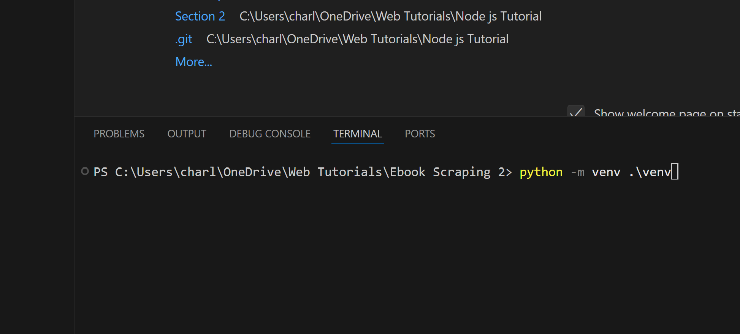
scrapy crawl quotes

**Tips:**

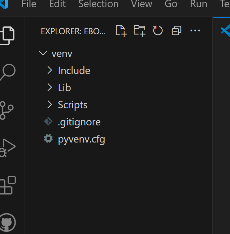
* **Selectors**: Use CSS or XPath selectors to extract data from HTML.
* **Pipelines**: Process and store the scraped data using item pipelines.
* **Middlewares**: Customiz

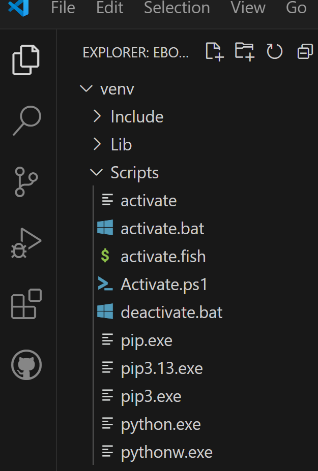
# Setting up the environment and Activate

Machine generated alternative text:
Sort 
View 
Name 
ebook scraping 
Ebook Scraping 2 
JavaScript Crash Course 
Luminar Tutorial 
Modern-HTML-CSS-From-The 
Node js Tutorial 
-Beginnin... 
Statu 
o 
o 
o First of all create a new folder, for this I will name Ebook Scraping 2

Then we need to create an environment

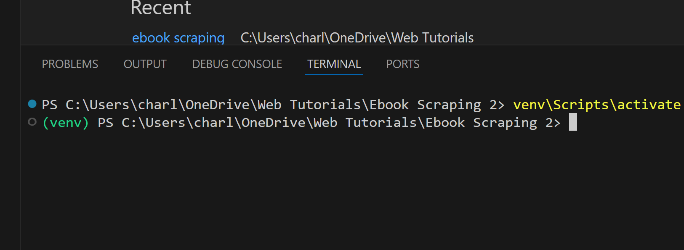
Using the code python -m venv .\venv to create an environment in the terminal

Once code is run you will see the new environment in the explorer bar



Inside the new environment venv there is sub the folder Scrips, and inside Scripts there is the file activate,

To activate the environment use the code in the terminal venv\Scripts\activate

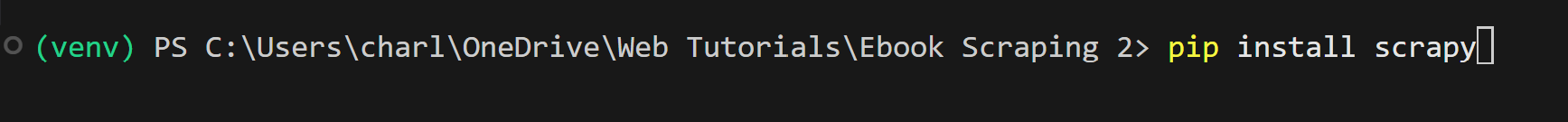


(venv) will now appear to show that you are in the new environment

# Create a Scrapy Project and install Scrapy

Before we create the project first we need to install scrapy using

pip install scrapy



Once scapy has been installed you can inspect by typing and you see all the options

scrapy

A screenshot of a computer program

AI-generated content may be incorrect.

Now that you have created an environment, we need to create a new project, with the name ebook\_scraper to do this we use the scrapy command utility

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.The new project ebook\_scraper will appear in the explorer bar

scrapy startproject ebook\_scraper

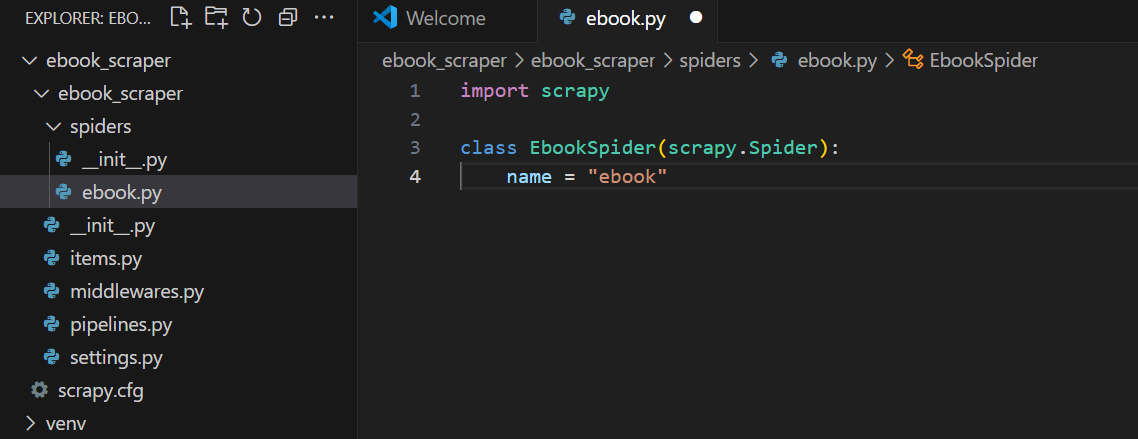
# Creating the Spider

A screenshot of a computer

AI-generated content may be incorrect.As you can see from the previous image that there is already a folder called spiders, already populated with some python tools to help our scraping, what we need to do now is create a python file that will contain the script that targets what information we intend to extract

Now we have created a new file within the spiders folder that we have named in this instance we will name it ebook, as ebook is what we are going to scrape

Next we need to create our own spider to do our scraping, this spider will contain the information of the web site that we want to scrape and the data targeted for extraction.



This spider will be able to do a lot of things like sending requests, handling the response and extracting the data

Lets break down the code

this imports the scrapy library.

Import scrapy

Scrapy then provides the class and other tools used for web scraping.

Spider

Class = EbookSpider(scrapey.Spider):

This declares a new class called

EbookSpider

Inherits from , the base class that all Scrapy spiders should extend.

Scrapy.Spider

Name = “ebook”

Sets the spider’s name. This is how you refer to it when running it from the command line:

# Sending the request

To send the request we need to let our spider know which website we would like it to crawl over and retrieve the data we would like, there are two methods available and that can be used to define where your scrapy spider begins crawling, but they differ on how they are used and how much control you want.

Start\_urls

Start\_requests()

✅ Key Differences Between and

Start\_urls

Start\_requests()

| **Feature** | **start\_urls** | **start\_requests()** |
| --- | --- | --- |
| **Type** | Class attribute (list of URLs) | Method |
| **Control** | Simple, automatic | Full control |
| **When to Use** | For simple spiders | When you need custom headers, POST requests, or logging |
| **Callback** | Always uses parse() | You can specify any callback |
| **Customization** | Limited | Full flexibility |

We can see this in the code example start request,

A screen shot of a computer program

AI-generated content may be incorrect.

Broken down

def start\_requests(self):

* This is the method Scrapy runs first when starting the spider.
* You define the list of URLs you want to start crawling from here.

🔹 What does self mean in def start\_requests(self):?

In Python, self refers to **the instance of the class** — the specific object created from that class. It's how the method accesses data and functions that belong to the same object.

**✅ Example:**

class EbookSpider(scrapy.Spider):

name = "ebook"

def start\_requests(self):

print(self.name) *# Access the 'name' attribute of this spider instance*

* Here, self.name refers to the name = "ebook" set at the class level.
* When Scrapy runs your spider, it creates an instance of EbookSpider.
* self is that instance — so you can access all its methods and attributes.

**🔁 In Simple Terms:**

* Think of self like **"this object"** in other languages (e.g., this in Java or JavaScript).
* It lets the method operate on the **specific spider object** being run.

**🔧 Why It's Needed:**

Python methods in classes must take self as the first argument so that:

* The method knows which object it’s operating on.
* It can access the spider’s internal data (like self.name, self.start\_urls, etc.).

urls = [……….]

A list of URLs the spider will start scraping, in this case we are only using one url

"https://books.toscrape.com"

for url in urls:

Loops through the URLs in the list.

yield scrapy.Request(url=url, callback=self.parse)

* For each URL, Scrapy creates an HTTP request.
* Once the response is received, it calls the parse method with that response.

def parse(self, response):

* This method handles the HTML response.
* You write your scraping logic here using XPath or CSS selectors to extract data.

pass

* A placeholder. Replace it with your real parsing code (e.g., extract titles, prices, etc.).

**✅ Key Differences Between start\_requests() and start\_urls**

| **Feature** | **start\_urls** | **start\_requests()** |
| --- | --- | --- |
| **Type** | Class attribute (list of URLs) | Method |
| **Control** | Simple, automatic | Full control |
| **When to Use** | For simple spiders | When you need custom headers, POST requests, or logging |
| **Callback** | Always uses parse() | You can specify any callback |
| **Customization** | Limited | Full flexibility |

**🔹 Using start\_urls (simple & automatic)**

python

class EbookSpider(scrapy.Spider):

name = "ebook"

start\_urls = [

"https://example.com/ebook1",

"https://example.com/ebook2"

]

def parse(self, response):

*# This is automatically called for each URL in* start\_urls

pass

Scrapy **automatically** generates requests from start\_urls.

* These requests will go to the default parse() method.
* Good for quick scrapers where you don’t need anything fancy.

**🔸 Using start\_requests() (custom & powerful)**

python

class EbookSpider(scrapy.Spider):

name = "ebook"

def start\_requests(self):

urls = [

"https://example.com/ebook1",

"https://example.com/ebook2"

]

for url in urls:

yield scrapy.Request(url=url, callback=self.parse)

* Lets you:
  + Add **custom headers**
  + Use **different callback methods**
  + Send **POST requests**
  + Log URLs or manipulate them before sending
* Required when you need fine-grained control over each request.

**✅ Summary**

* Use start\_urls when you just want to scrape a static list of URLs using parse().
* Use start\_requests() when you need **more control** over each request (e.g., dynamic headers, session management, or advanced logic).

Would you like an example of start\_requests() with custom headers or POST data?

4o

**You said:**

what dose self in the syntext "def start\_requests(self):" signify

**ChatGPT said:**

Great question — understanding self is fundamental to working with Python classes, including Scrapy spiders.

**🔹 What does self mean in def start\_requests(self):?**

In Python, self refers to **the instance of the class** — the specific object created from that class. It's how the method a

ccesses data and functions that belong to the same object.

**✅ Example:**

Python

class EbookSpider(scrapy.Spider):

name = "ebook"

def start\_requests(self):

print(self.name) *# Access the 'name' attribute of this spider instance*

* Here, self.name refers to the name = "ebook" set at the class level.
* When Scrapy runs your spider, it creates an instance of EbookSpider.
* self is that instance — so you can access all its methods and attributes.

.

**🔁 In Simple Terms:**

* Think of self like **"this object"** in other languages (e.g., this in Java or JavaScript).
* It lets the method operate on the **specific spider object** being run.

Python methods in classes must take self as the first argument so that:

* The method knows which object it’s operating on.
* It can access the spider’s internal data (like self.name, self.start\_urls, etc.).

topics-spiders:

=============

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()](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider.start_requests" \o "scrapy.Spider.start_requests)** method which (by default) generates **Request** for the URLs specified in the **[start\_urls](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider.start_urls" \o "scrapy.Spider.start_urls)** and the [**parse**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.parse) method as callback function for the Requests.

1. In the callback function, you parse the response (web page) and return [item objects](https://docs.scrapy.org/en/latest/topics/items.html#topics-items), **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.
2. In callback functions, you parse the page contents, typically using [Selectors](https://docs.scrapy.org/en/latest/topics/selectors.html#topics-selectors) (but you can also use BeautifulSoup, lxml or whatever mechanism you prefer) and generate items with the parsed data.
3. Finally, the items returned from the spider will be typically persisted to a database (in some [Item Pipeline](https://docs.scrapy.org/en/latest/topics/item-pipeline.html#topics-item-pipeline)) or written to a file using [Feed exports](https://docs.scrapy.org/en/latest/topics/feed-exports.html#topics-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

classscrapy.spiders.Spider**[¶](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.spiders.Spider" \o "Permalink to this definition)**

*class*scrapy.Spider**[¶](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider" \o "Permalink to this definition)**

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()](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider.start_requests" \o "scrapy.Spider.start_requests) implementation which sends requests from the [start\_urls](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider.start_urls" \o "scrapy.Spider.start_urls) spider attribute and calls the spider’s method parse for each of the resulting responses.

name[**¶**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.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](https://en.wikipedia.org/wiki/Top-level_domain). So, for example, a spider that crawls mywebsite.com would often be called mywebsite.

allowed\_domains**[¶](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider.allowed_domains" \o "Permalink to this definition)**

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](https://docs.scrapy.org/en/latest/topics/downloader-middleware.html" \l "scrapy.downloadermiddlewares.offsite.OffsiteMiddleware" \o "scrapy.downloadermiddlewares.offsite.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[¶](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider.start_urls" \o "Permalink to this definition)**

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 **Request** will be generated successively from data contained in the start URLs.

**custom\_settings[¶](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider.custom_settings" \o "Permalink to this definition)**

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](https://docs.scrapy.org/en/latest/topics/settings.html#topics-settings-ref).

**crawler**[**¶**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.crawler)

This attribute is set by the **[from\_crawler()](https://docs.scrapy.org/en/latest/topics/item-pipeline.html" \l "from_crawler" \o "from_crawler)** class method after initializing the class, and links to the [**Crawler**](https://docs.scrapy.org/en/latest/topics/api.html#scrapy.crawler.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](https://docs.scrapy.org/en/latest/topics/api.html#topics-api-crawler) to know more about them.

**settings**[**¶**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.settings)

Configuration for running this spider. This is a [**Settings**](https://docs.scrapy.org/en/latest/topics/api.html#scrapy.settings.Settings) instance, see the [Settings](https://docs.scrapy.org/en/latest/topics/settings.html#topics-settings) topic for a detailed introduction on this subject.

**logger**[**¶**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.logger)

Python logger created with the Spider’s [**name**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.name). You can use it to send log messages through it as described on [Logging from Spiders](https://docs.scrapy.org/en/latest/topics/logging.html#topics-logging-from-spiders).

**state**[**¶**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.state)

A dict you can use to persist some spider state between batches. See [Keeping persistent state between batches](https://docs.scrapy.org/en/latest/topics/jobs.html#topics-keeping-persistent-state-between-batches) to know more about it.

**from\_crawler(*crawler*, *\*args*, *\*\*kwargs*)**[**¶**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.from_crawler)

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\_\_()**](https://docs.scrapy.org/en/latest/topics/feed-exports.html#init__) method, calling it with the given arguments args and named arguments kwargs.

Nonetheless, this method sets the [**crawler**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.crawler) and [**settings**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.settings) attributes in the new instance so they can be accessed later inside the spider’s code.

*Changed in version 2.11:*The settings in crawler.settings can now be modified in this method, which is handy if you want to modify them based on arguments. As a consequence, these settings aren’t the final values as they can be modified later by e.g. [add-ons](https://docs.scrapy.org/en/latest/topics/addons.html#topics-addons). For the same reason, most of the [**Crawler**](https://docs.scrapy.org/en/latest/topics/api.html#scrapy.crawler.Crawler) attributes aren’t initialized at this point.

The final settings and the initialized [**Crawler**](https://docs.scrapy.org/en/latest/topics/api.html#scrapy.crawler.Crawler) attributes are available in the **[start\_requests()](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider.start_requests" \o "scrapy.Spider.start_requests)** method, handlers of the **[engine\_started](https://docs.scrapy.org/en/latest/topics/signals.html" \l "std-signal-engine_started)** signal and later.

**Parameters:**

* **crawler** ([**Crawler**](https://docs.scrapy.org/en/latest/topics/api.html#scrapy.crawler.Crawler) instance) – crawler to which the spider will be bound
* **args** ([*list*](https://docs.python.org/3/library/stdtypes.html#list)) – arguments passed to the [**\_\_init\_\_()**](https://docs.scrapy.org/en/latest/topics/feed-exports.html#init__) method
* **kwargs** (*[dict](https://docs.python.org/3/library/stdtypes.html" \l "dict" \o "(in Python v3.13))*) – keyword arguments passed to the [**\_\_init\_\_()**](https://docs.scrapy.org/en/latest/topics/feed-exports.html#init__) method

***classmethod*update\_settings(*settings*)**[**¶**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider.update_settings)

The update\_settings() method is used to modify the spider’s settings and is called during initialization of a spider instance.

It takes a [**Settings**](https://docs.scrapy.org/en/latest/topics/api.html#scrapy.settings.Settings) object as a parameter and can add or update the spider’s configuration values. This method is a class method, meaning that it is called on the [**Spider**](https://docs.scrapy.org/en/latest/topics/spiders.html#scrapy.Spider) class and allows all instances of the spider to share the same configuration.

While per-spider settings can be set in **[custom\_settings](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider.custom_settings" \o "scrapy.Spider.custom_settings)**, using update\_settings() allows you to dynamically add, remove or change settings based on other settings, spider attributes or other factors and use setting priorities other than 'spider'. Also, it’s easy to extend update\_settings() in a subclass by overriding it, while doing the same with **[custom\_settings](https://docs.scrapy.org/en/latest/topics/spiders.html" \l "scrapy.Spider.custom_settings" \o "scrapy.Spider.custom_settings)** can be hard.

For example, suppose a spider needs to modify [**FEEDS**](https://docs.scrapy.org/en/latest/topics/feed-exports.html#std-setting-FEEDS):

class:: scrapy.spiders.Spider

class:: scrapy.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 :meth:`start\_requests` implementation which sends requests from

the :attr:`start\_urls` spider attribute and calls the spider's method ``parse``

for each of the resulting responses.

.. attribute:: 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``.

.. attribute:: 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

:class:`~scrapy.downloadermiddlewares.offsite.OffsiteMiddleware` is

enabled.

Let's say your target url is ``https://www.example.com/1.html``,

then add ``'example.com'`` to the list.

.. attribute:: 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 :class:`~scrapy.Request` will be generated successively from data

contained in the start URLs.

.. attribute:: 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:

:ref:`topics-settings-ref`.

.. attribute:: crawler

This attribute is set by the :meth:`from\_crawler` class method after

initializing the class, and links to the

:class:`~scrapy.crawler.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 :ref:`topics-api-crawler` to know more about them.

.. attribute:: settings

Configuration for running this spider. This is a

:class:`~scrapy.settings.Settings` instance, see the

:ref:`topics-settings` topic for a detailed introduction on this subject.

.. attribute:: logger

Python logger created with the Spider's :attr:`name`. You can use it to

send log messages through it as described on

:ref:`topics-logging-from-spiders`.

.. attribute:: state

A dict you can use to persist some spider state between batches.

See :ref:`topics-keeping-persistent-state-between-batches` to know more about it.

.. method:: 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 :meth:`\_\_init\_\_` method, calling

it with the given arguments ``args`` and named arguments ``kwargs``.

Nonetheless, this method sets the :attr:`crawler` and :attr:`settings`

attributes in the new instance so they can be accessed later inside the

spider's code.

.. versionchanged:: 2.11

The settings in ``crawler.settings`` can now be modified in this

method, which is handy if you want to modify them based on

arguments. As a consequence, these settings aren't the final values

as they can be modified later by e.g. :ref:`add-ons

<topics-addons>`. For the same reason, most of the

:class:`~scrapy.crawler.Crawler` attributes aren't initialized at

this point.

The final settings and the initialized

:class:`~scrapy.crawler.Crawler` attributes are available in the

:meth:`start\_requests` method, handlers of the

:signal:`engine\_started` signal and later.

:param crawler: crawler to which the spider will be bound

:type crawler: :class:`~scrapy.crawler.Crawler` instance

:param args: arguments passed to the :meth:`\_\_init\_\_` method

:type args: list

:param kwargs: keyword arguments passed to the :meth:`\_\_init\_\_` method

:type kwargs: dict

.. classmethod:: update\_settings(settings)

The ``update\_settings()`` method is used to modify the spider's settings

and is called during initialization of a spider instance.

It takes a :class:`~scrapy.settings.Settings` object as a parameter and

can add or update the spider's configuration values. This method is a

class method, meaning that it is called on the :class:`~scrapy.Spider`

class and allows all instances of the spider to share the same

configuration.

While per-spider settings can be set in

:attr:`~scrapy.Spider.custom\_settings`, using ``update\_settings()``

allows you to dynamically add, remove or change settings based on other

settings, spider attributes or other factors and use setting priorities

other than ``'spider'``. Also, it's easy to extend ``update\_settings()``

in a subclass by overriding it, while doing the same with

:attr:`~scrapy.Spider.custom\_settings` can be hard.

For example, suppose a spider needs to modify :setting:`FEEDS`:

.. code-block:: python

import scrapy

class MySpider(scrapy.Spider):

name = "myspider"

custom\_feed = {

"/home/user/documents/items.json": {

"format": "json",

"indent": 4,

}

}

@classmethod

def update\_settings(cls, settings):

super().update\_settings(settings)

settings.setdefault("FEEDS", {}).update(cls.custom\_feed)

.. method:: start\_requests()

This method must return an iterable with the first Requests to crawl and/or with :ref:`item objects

<topics-items>` 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

:meth:`start\_requests` as a generator.

The default implementation generates ``Request(url, dont\_filter=True)``

for each url in :attr:`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:

.. code-block:: python

import scrapy

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

.. method:: 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 :class:`Spider` class.

This method, as well as any other Request callback, must return a

:class:`~scrapy.Request` object, an :ref:`item object <topics-items>`, an

iterable of :class:`~scrapy.Request` objects and/or :ref:`item objects

<topics-items>`, or ``None``.

:param response: the response to parse

:type response: :class:`~scrapy.http.Response`

.. method:: log(message, [level, component])

Wrapper that sends a log message through the Spider's :attr:`logger`,

kept for backward compatibility. For more information see

:ref:`topics-logging-from-spiders`.

.. method:: closed(reason)

Called when the spider closes. This method provides a shortcut to

signals.connect() for the :signal:`spider\_closed` signal.

Let's see an example:

.. code-block:: python

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:

.. code-block:: python

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").getall():

yield {"title": h3}

for href in response.xpath("//a/@href").getall():

yield scrapy.Request(response.urljoin(href), self.parse)

Instead of :attr:`~.start\_urls` you can use :meth:`~.start\_requests` directly;

to give data more structure you can use :class:`~scrapy.Item` objects:

.. skip: next

.. code-block:: python

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").getall():

yield MyItem(title=h3)

for href in response.xpath("//a/@href").getall():

yield scrapy.Request(response.urljoin(href), self.parse)

.. \_spiderargs:

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 :command:`crawl` command using the

``-a`` option. For example::

scrapy crawl myspider -a category=electronics

Spiders can access arguments in their `\_\_init\_\_` methods:

.. code-block:: python

import scrapy

class MySpider(scrapy.Spider):

name = "myspider"

def \_\_init\_\_(self, category=None, \*args, \*\*kwargs):

super(MySpider, self).\_\_init\_\_(\*args, \*\*kwargs)

self.start\_urls = [f"http://www.example.com/categories/{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:

.. code-block:: python

import scrapy

class MySpider(scrapy.Spider):

name = "myspider"

def start\_requests(self):

yield scrapy.Request(f"http://www.example.com/categories/{self.category}")

If you are :ref:`running Scrapy from a script <run-from-script>`, you can

specify spider arguments when calling

:class:`CrawlerProcess.crawl <scrapy.crawler.CrawlerProcess.crawl>` or

:class:`CrawlerRunner.crawl <scrapy.crawler.CrawlerRunner.crawl>`:

.. skip: next

.. code-block:: python

process = CrawlerProcess()

process.crawl(MySpider, category="electronics")

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 :func:`ast.literal\_eval` or :func:`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 :class:`~scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware`

or the user agent

used by :class:`~scrapy.downloadermiddlewares.useragent.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`\_.

.. \_builtin-spiders:

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:

.. code-block:: python

import scrapy

class TestItem(scrapy.Item):

id = scrapy.Field()

name = scrapy.Field()

description = scrapy.Field()

.. currentmodule:: scrapy.spiders

CrawlSpider

-----------

.. class:: 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:

.. attribute:: rules

Which is a list of one (or more) :class:`Rule` objects. Each :class:`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 overridable method:

.. method:: parse\_start\_url(response, \*\*kwargs)

This method is called for each response produced for the URLs in

the spider's ``start\_urls`` attribute. It allows to parse

the initial responses and must return either an

:ref:`item object <topics-items>`, a :class:`~scrapy.Request`

object, or an iterable containing any of them.

Crawling rules

~~~~~~~~~~~~~~

.. autoclass:: Rule

``link\_extractor`` is a :ref:`Link Extractor <topics-link-extractors>` object which

defines how links will be extracted from each crawled page. Each produced link will

be used to generate a :class:`~scrapy.Request` object, which will contain the

link's text in its ``meta`` dictionary (under the ``link\_text`` key).

If omitted, a default link extractor created with no arguments will be used,

resulting in all links being extracted.

``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 :class:`~scrapy.http.Response`

as its first argument and must return either a single instance or an iterable of

:ref:`item objects <topics-items>` and/or :class:`~scrapy.Request` objects

(or any subclass of them). As mentioned above, the received :class:`~scrapy.http.Response`

object will contain the text of the link that produced the :class:`~scrapy.Request`

in its ``meta`` dictionary (under the ``link\_text`` key)

``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 for every

:class:`~scrapy.Request` extracted by this rule. This callable should

take said request as first argument and the :class:`~scrapy.http.Response`

from which the request originated as second argument. It must return a

``Request`` object or ``None`` (to filter out the request).

``errback`` is a callable or a string (in which case a method from the spider

object with that name will be used) to be called if any exception is

raised while processing a request generated by the rule.

It receives a :class:`Twisted Failure <twisted.python.failure.Failure>`

instance as first parameter.

.. warning:: Because of its internal implementation, you must explicitly set

callbacks for new requests when writing :class:`CrawlSpider`-based spiders;

unexpected behaviour can occur otherwise.

.. versionadded:: 2.0

The \*errback\* parameter.

CrawlSpider example

~~~~~~~~~~~~~~~~~~~

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

.. code-block:: python

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=(r"category\.php",), deny=(r"subsection\.php",))),

# Extract links matching 'item.php' and parse them with the spider's method parse\_item

Rule(LinkExtractor(allow=(r"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()').get()

item["description"] = response.xpath(

'//td[@id="item\_description"]/text()'

).get()

item["link\_text"] = response.meta["link\_text"]

url = response.xpath('//td[@id="additional\_data"]/@href').get()

return response.follow(

url, self.parse\_additional\_page, cb\_kwargs=dict(item=item)

)

def parse\_additional\_page(self, response, item):

item["additional\_data"] = response.xpath(

'//p[@id="additional\_data"]/text()'

).get()

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 :class:`~scrapy.Item` will be filled with it.

XMLFeedSpider

-------------

.. class:: 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:

.. attribute:: 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 :class:`~scrapy.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 :class:`~scrapy.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'``.

.. attribute:: itertag

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

itertag = 'product'

.. attribute:: 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

:meth:`~scrapy.Selector.register\_namespace` method.

You can then specify nodes with namespaces in the :attr:`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 overridable

methods too:

.. method:: 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).

.. method:: parse\_node(response, selector)

This method is called for the nodes matching the provided tag name

(``itertag``). Receives the response and an

:class:`~scrapy.Selector` for each node. Overriding this

method is mandatory. Otherwise, you spider won't work. This method

must return an :ref:`item object <topics-items>`, a

:class:`~scrapy.Request` object, or an iterable containing any of

them.

.. method:: 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).

.. warning:: Because of its internal implementation, you must explicitly set

callbacks for new requests when writing :class:`XMLFeedSpider`-based spiders;

unexpected behaviour can occur otherwise.

XMLFeedSpider example

~~~~~~~~~~~~~~~~~~~~~

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

.. skip: next

.. code-block:: python

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.getall())

)

item = TestItem()

item["id"] = node.xpath("@id").get()

item["name"] = node.xpath("name").get()

item["description"] = node.xpath("description").get()

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 :class:`~scrapy.Item`.

CSVFeedSpider

-------------

.. class:: 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 :meth:`parse\_row`.

.. attribute:: delimiter

A string with the separator character for each field in the CSV file

Defaults to ``','`` (comma).

.. attribute:: quotechar

A string with the enclosure character for each field in the CSV file

Defaults to ``'"'`` (quotation mark).

.. attribute:: headers

A list of the column names in the CSV file.

.. method:: 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

:class:`CSVFeedSpider`:

.. skip: next

.. code-block:: python

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:: 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`\_.

.. attribute:: 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.

.. attribute:: 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.

.. attribute:: sitemap\_follow

A list of regexes of sitemap that should be followed. This is only

for sites that use `Sitemap index files`\_ that point to other sitemap

files.

By default, all sitemaps are followed.

.. attribute:: 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.

.. method:: sitemap\_filter(entries)

This is a filter function that could be overridden to select sitemap entries

based on their attributes.

For example::

<url>

<loc>http://example.com/</loc>

<lastmod>2005-01-01</lastmod>

</url>

We can define a ``sitemap\_filter`` function to filter ``entries`` by date:

.. code-block:: python

from datetime import datetime

from scrapy.spiders import SitemapSpider

class FilteredSitemapSpider(SitemapSpider):

name = "filtered\_sitemap\_spider"

allowed\_domains = ["example.com"]

sitemap\_urls = ["http://example.com/sitemap.xml"]

def sitemap\_filter(self, entries):

for entry in entries:

date\_time = datetime.strptime(entry["lastmod"], "%Y-%m-%d")

if date\_time.year >= 2005:

yield entry

This would retrieve only ``entries`` modified on 2005 and the following

years.

Entries are dict objects extracted from the sitemap document.

Usually, the key is the tag name and the value is the text inside it.

It's important to notice that:

- as the loc attribute is required, entries without this tag are discarded

- alternate links are stored in a list with the key ``alternate``

(see ``sitemap\_alternate\_links``)

- namespaces are removed, so lxml tags named as ``{namespace}tagname`` become only ``tagname``

If you omit this method, all entries found in sitemaps will be

processed, observing other attributes and their settings.

SitemapSpider examples

~~~~~~~~~~~~~~~~~~~~~~

Simplest example: process all urls discovered through sitemaps using the

``parse`` callback:

.. code-block:: python

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:

.. code-block:: python

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``:

.. code-block:: python

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:

.. code-block:: python

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 ...

.. \_Sitemaps: https://www.sitemaps.org/index.html

.. \_Sitemap index files: https://www.sitemaps.org/protocol.html#index

.. \_robots.txt: https://www.robotstxt.org/

.. \_TLD: https://en.wikipedia.org/wiki/Top-level\_domain

.. \_Scrapyd documentation: https://scrapyd.readthedocs.io/en/latest/