

---

# **scrapy Documentation**

***Release 1.5***

**terryxi**

**Dec 11, 2018**



---

---

<b>1</b>		<b>3</b>
<b>2</b>		<b>5</b>
2.1	Scrapy . . . . .	5
2.2	. . . . .	7
2.3	Scrapy . . . . .	11
2.4	Examples . . . . .	22
<b>3</b>	<b>Basic concepts</b>	<b>23</b>
3.1	Command line tool . . . . .	23
3.2	Spiders . . . . .	32
3.3	Selectors . . . . .	42
3.4	Items . . . . .	54
3.5	Item Loaders . . . . .	58
3.6	Scrapy shell . . . . .	68
3.7	Item Pipeline . . . . .	73
3.8	Feed exports . . . . .	76
3.9	Requests and Responses . . . . .	81
3.10	Link Extractors . . . . .	91
3.11	Settings . . . . .	93
3.12	Exceptions . . . . .	111
<b>4</b>	<b>Built-in services</b>	<b>115</b>
4.1	Logging . . . . .	115
4.2	Stats Collection . . . . .	118
4.3	Sending e-mail . . . . .	120
4.4	Telnet Console . . . . .	122
4.5	Web Service . . . . .	124
<b>5</b>	<b>Solving specific problems</b>	<b>127</b>
5.1	Frequently Asked Questions . . . . .	127
5.2	Debugging Spiders . . . . .	132
5.3	Spiders Contracts . . . . .	134
5.4	Common Practices . . . . .	136
5.5	Broad Crawls . . . . .	140
5.6	Using your browser's Developer Tools for scraping . . . . .	142
5.7	Debugging memory leaks . . . . .	147
5.8	Downloading and processing files and images . . . . .	152

---

5.9	Deploying Spiders . . . . .	159
5.10	AutoThrottle extension . . . . .	159
5.11	Benchmarking . . . . .	162
5.12	Jobs: pausing and resuming crawls . . . . .	163
<b>6</b>	<b>Extending Scrapy</b>	<b>167</b>
6.1	Architecture overview . . . . .	167
6.2	Downloader Middleware . . . . .	170
6.3	Spider Middleware . . . . .	183
6.4	Extensions . . . . .	188
6.5	Core API . . . . .	194
6.6	Signals . . . . .	197
6.7	Item Exporters . . . . .	201
	<b>Python Module Index</b>	<b>209</b>

Scrapy



# CHAPTER 1

---

---

- Try the [FAQ](#) – it’s got answers to some common questions.
- Looking for specific information? Try the [genindex](#) or [modindex](#).
- Ask or search questions in [StackOverflow](#) using the [scrapy](#) tag.
- Ask or search questions in the [Scrapy](#) subreddit.
- Search for questions on the archives of the [scrapy-users mailing list](#).
- Ask a question in the [#scrapy IRC channel](#),
- Report bugs with Scrapy in our [issue tracker](#).





## 2.1 Scrapy

Scrapyweb

Even though Scrapy was originally designed for [web scraping](#), it can also be used to extract data using APIs (such as [Amazon Associates Web Services](#)) or as a general purpose web crawler.

### 2.1.1 Walk-through of an example spider

In order to show you what Scrapy brings to the table, we'll walk you through an example of a Scrapy Spider using the simplest way to run a spider.

Here's the code for a spider that scrapes famous quotes from website <http://quotes.toscrape.com>, following the pagination:

```
import scrapy

class QuotesSpider(scrapy.Spider):
    name = "quotes"
    start_urls = [
        'http://quotes.toscrape.com/tag/humor/',
    ]

    def parse(self, response):
        for quote in response.css('div.quote'):
            yield {
                'text': quote.css('span.text::text').extract_first(),
                'author': quote.xpath('span/small/text()').extract_first(),
            }

        next_page = response.css('li.next a::attr("href")').extract_first()
        if next_page is not None:
            yield response.follow(next_page, self.parse)
```

Put this in a text file, name it to something like `quotes_spider.py` and run the spider using the **runspider** command:

```
scrapy runspider quotes_spider.py -o quotes.json
```

When this finishes you will have in the `quotes.json` file a list of the quotes in JSON format, containing text and author, looking like this (reformatted here for better readability):

```
[{
  "author": "Jane Austen",
  "text": "\u201cThe person, be it gentleman or lady, who has not pleasure in a_
\u2192good novel, must be intolerably stupid.\u201d"
},
{
  "author": "Groucho Marx",
  "text": "\u201cOutside of a dog, a book is man's best friend. Inside of a dog it
\u2192's too dark to read.\u201d"
},
{
  "author": "Steve Martin",
  "text": "\u201cA day without sunshine is like, you know, night.\u201d"
},
...]
```

### What just happened?

When you ran the command `scrapy runspider quotes_spider.py`, Scrapy looked for a Spider definition inside it and ran it through its crawler engine.

The crawl started by making requests to the URLs defined in the `start_urls` attribute (in this case, only the URL for quotes in *humor* category) and called the default callback method `parse`, passing the response object as an argument. In the `parse` callback, we loop through the quote elements using a CSS Selector, yield a Python dict with the extracted quote text and author, look for a link to the next page and schedule another request using the same `parse` method as callback.

Here you notice one of the main advantages about Scrapy: requests are *scheduled and processed asynchronously*. This means that Scrapy doesn't need to wait for a request to be finished and processed, it can send another request or do other things in the meantime. This also means that other requests can keep going even if some request fails or an error happens while handling it.

While this enables you to do very fast crawls (sending multiple concurrent requests at the same time, in a fault-tolerant way) Scrapy also gives you control over the politeness of the crawl through *a few settings*. You can do things like setting a download delay between each request, limiting amount of concurrent requests per domain or per IP, and even *using an auto-throttling extension* that tries to figure out these automatically.

---

**Note:** This is using *feed exports* to generate the JSON file, you can easily change the export format (XML or CSV, for example) or the storage backend (FTP or *Amazon S3*, for example). You can also write an *item pipeline* to store the items in a database.

---

### 2.1.2 What else?

You've seen how to extract and store items from a website using Scrapy, but this is just the surface. Scrapy provides a lot of powerful features for making scraping easy and efficient, such as:

- Built-in support for *selecting and extracting* data from HTML/XML sources using extended CSS selectors and XPath expressions, with helper methods to extract using regular expressions.
- An *interactive shell console* (IPython aware) for trying out the CSS and XPath expressions to scrape data, very useful when writing or debugging your spiders.
- Built-in support for *generating feed exports* in multiple formats (JSON, CSV, XML) and storing them in multiple backends (FTP, S3, local filesystem)
- Robust encoding support and auto-detection, for dealing with foreign, non-standard and broken encoding declarations.
- *Strong extensibility support*, allowing you to plug in your own functionality using *signals* and a well-defined API (middlewares, *extensions*, and *pipelines*).
- Wide range of built-in extensions and middlewares for handling:
  - cookies and session handling
  - HTTP features like compression, authentication, caching
  - user-agent spoofing
  - robots.txt
  - crawl depth restriction
  - and more
- A *Telnet console* for hooking into a Python console running inside your Scrapy process, to introspect and debug your crawler
- Plus other goodies like reusable spiders to crawl sites from *Sitemaps* and XML/CSV feeds, a media pipeline for *automatically downloading images* (or any other media) associated with the scraped items, a caching DNS resolver, and much more!

### 2.1.3 What's next?

The next steps for you are to *install Scrapy*, *follow through the tutorial* to learn how to create a full-blown Scrapy project and *join the community*. Thanks for your interest!

## 2.2

### 2.2.1 Scrapy

Scrapy Python 2.7 Python 3.4 under CPython (default Python implementation) and PyPy (starting with PyPy 5.9).

If you're using *Anaconda* or *Miniconda*, you can install the package from the *conda-forge* channel, which has up-to-date packages for Linux, Windows and OS X.

To install Scrapy using *conda*, run:

```
conda install -c conda-forge scrapy
```

Alternatively, if you're already familiar with installation of Python packages, you can install Scrapy and its dependencies from *PyPI* with:

```
pip install Scrapy
```

Note that sometimes this may require solving compilation issues for some Scrapy dependencies depending on your operating system, so be sure to check the [Platform specific installation notes](#).

We strongly recommend that you install Scrapy in a [dedicated virtualenv](#), to avoid conflicting with your system packages.

For more detailed and platform specifics instructions, read on.

### Things that are good to know

Scrapy is written in pure Python and depends on a few key Python packages (among others):

- [lxml](#), an efficient XML and HTML parser
- [parsel](#), an HTML/XML data extraction library written on top of [lxml](#),
- [w3lib](#), a multi-purpose helper for dealing with URLs and web page encodings
- [twisted](#), an asynchronous networking framework
- [cryptography](#) and [pyOpenSSL](#), to deal with various network-level security needs

The minimal versions which Scrapy is tested against are:

- Twisted 14.0
- [lxml](#) 3.4
- [pyOpenSSL](#) 0.14

Scrapy may work with older versions of these packages but it is not guaranteed it will continue working because it's not being tested against them.

Some of these packages themselves depends on non-Python packages that might require additional installation steps depending on your platform. Please check [platform-specific guides below](#).

In case of any trouble related to these dependencies, please refer to their respective installation instructions:

- [lxml installation](#)
- [cryptography installation](#)

### Using a virtual environment (recommended)

TL;DR: We recommend installing Scrapy inside a virtual environment on all platforms.

Python packages can be installed either globally (a.k.a system wide), or in user-space. We do not recommend installing scrapy system wide.

Instead, we recommend that you install scrapy within a so-called “virtual environment” ([virtualenv](#)). Virtualenvs allow you to not conflict with already-installed Python system packages (which could break some of your system tools and scripts), and still install packages normally with `pip` (without `sudo` and the likes).

To get started with virtual environments, see [virtualenv installation instructions](#). To install it globally (having it globally installed actually helps here), it should be a matter of running:

```
$ [sudo] pip install virtualenv
```

Check this [user guide](#) on how to create your virtualenv.

---

**Note:** If you use Linux or OS X, [virtualenvwrapper](#) is a handy tool to create virtualenvs.

---

Once you have created a virtualenv, you can install scrapy inside it with `pip`, just like any other Python package. (See [platform-specific guides](#) below for non-Python dependencies that you may need to install beforehand).

Python virtualenvs can be created to use Python 2 by default, or Python 3 by default.

- If you want to install scrapy with Python 3, install scrapy within a Python 3 virtualenv.
- And if you want to install scrapy with Python 2, install scrapy within a Python 2 virtualenv.

## 2.2.2 Platform specific installation notes

### Windows

Though it's possible to install Scrapy on Windows using `pip`, we recommend you to install [Anaconda](#) or [Miniconda](#) and use the package from the [conda-forge](#) channel, which will avoid most installation issues.

Once you've installed [Anaconda](#) or [Miniconda](#), install Scrapy with:

```
conda install -c conda-forge scrapy
```

### Ubuntu 14.04 or above

Scrapy is currently tested with recent-enough versions of `lxml`, `twisted` and `pyOpenSSL`, and is compatible with recent Ubuntu distributions. But it should support older versions of Ubuntu too, like Ubuntu 14.04, albeit with potential issues with TLS connections.

**Don't** use the `python-scrapy` package provided by Ubuntu, they are typically too old and slow to catch up with latest Scrapy.

To install scrapy on Ubuntu (or Ubuntu-based) systems, you need to install these dependencies:

```
sudo apt-get install python-dev python-pip libxml2-dev libxslt1-dev zlib1g-dev libffi-  
↳dev libssl-dev
```

- `python-dev`, `zlib1g-dev`, `libxml2-dev` and `libxslt1-dev` are required for `lxml`
- `libssl-dev` and `libffi-dev` are required for cryptography

If you want to install scrapy on Python 3, you'll also need Python 3 development headers:

```
sudo apt-get install python3 python3-dev
```

Inside a [virtualenv](#), you can install Scrapy with `pip` after that:

```
pip install scrapy
```

---

**Note:** The same non-Python dependencies can be used to install Scrapy in Debian Jessie (8.0) and above.

---

### Mac OS X

Building Scrapy's dependencies requires the presence of a C compiler and development headers. On OS X this is typically provided by Apple's Xcode development tools. To install the Xcode command line tools open a terminal window and run:

```
xcode-select --install
```

There's a [known issue](#) that prevents `pip` from updating system packages. This has to be addressed to successfully install Scrapy and its dependencies. Here are some proposed solutions:

- *(Recommended)* **Don't** use system python, install a new, updated version that doesn't conflict with the rest of your system. Here's how to do it using the [homebrew](#) package manager:

- Install [homebrew](#) following the instructions in <https://brew.sh/>
- Update your `PATH` variable to state that homebrew packages should be used before system packages (Change `.bashrc` to `.zshrc` accordingly if you're using `zsh` as default shell):

```
echo "export PATH=/usr/local/bin:/usr/local/sbin:$PATH" >> ~/.bashrc
```

- Reload `.bashrc` to ensure the changes have taken place:

```
source ~/.bashrc
```

- Install python:

```
brew install python
```

- Latest versions of python have `pip` bundled with them so you won't need to install it separately. If this is not the case, upgrade python:

```
brew update; brew upgrade python
```

- *(Optional)* Install Scrapy inside an isolated python environment.

This method is a workaround for the above OS X issue, but it's an overall good practice for managing dependencies and can complement the first method.

[virtualenv](#) is a tool you can use to create virtual environments in python. We recommended reading a tutorial like <http://docs.python-guide.org/en/latest/dev/virtualenvs/> to get started.

After any of these workarounds you should be able to install Scrapy:

```
pip install Scrapy
```

## PyPy

We recommend using the latest PyPy version. The version tested is 5.9.0. For PyPy3, only Linux installation was tested.

Most scrapy dependencies now have binary wheels for CPython, but not for PyPy. This means that these dependencies will be built during installation. On OS X, you are likely to face an issue with building Cryptography dependency, solution to this problem is described [here](#), that is to `brew install openssl` and then export the flags that this command recommends (only needed when installing scrapy). Installing on Linux has no special issues besides installing build dependencies. Installing scrapy with PyPy on Windows is not tested.

You can check that scrapy is installed correctly by running `scrapy bench`. If this command gives errors such as `TypeError: ... got 2 unexpected keyword arguments`, this means that `setuptools` was unable to pick up one PyPy-specific dependency. To fix this issue, run `pip install 'PyPyDispatcher>=2.1.0'`.

## 2.3 Scrapy

In this tutorial, we'll assume that Scrapy is already installed on your system. If that's not the case, see .

Scrapy .

We are going to scrape [quotes.toscrape.com](https://quotes.toscrape.com), a website that lists quotes from famous authors.

This tutorial will walk you through these tasks:

1. Creating a new Scrapy project
2. Writing a *spider* to crawl a site and extract data
3. Exporting the scraped data using the command line
4. Changing spider to recursively follow links
5. Using spider arguments

Scrapy is written in [Python](#). If you're new to the language you might want to start by getting an idea of what the language is like, to get the most out of Scrapy.

If you're already familiar with other languages, and want to learn Python quickly, we recommend reading through [Dive Into Python 3](#). Alternatively, you can follow the [Python Tutorial](#).

If you're new to programming and want to start with Python, the following books may be useful to you:

- [Automate the Boring Stuff With Python](#)
- [How To Think Like a Computer Scientist](#)
- [Learn Python 3 The Hard Way](#)

You can also take a look at [this list of Python resources for non-programmers](#), as well as the [suggested resources in the learnpython-subreddit](#).

### 2.3.1 Creating a project

Before you start scraping, you will have to set up a new Scrapy project. Enter a directory where you'd like to store your code and run:

```
scrapy startproject tutorial
```

This will create a `tutorial` directory with the following contents:

```
tutorial/
  scrapy.cfg          # deploy configuration file

  tutorial/           # project's Python module, you'll import your code from here
    __init__.py

    items.py          # project items definition file

    middlewares.py     # project middlewares file

    pipelines.py       # project pipelines file

    settings.py        # project settings file
```

(continues on next page)

(continued from previous page)

```
spiders/          # a directory where you'll later put your spiders
__init__.py
```

## 2.3.2 Our first Spider

Spiders are classes that you define and that Scrapy uses to scrape information from a website (or a group of websites). They must subclass `scrapy.Spider` and define the initial requests to make, optionally how to follow links in the pages, and how to parse the downloaded page content to extract data.

This is the code for our first Spider. Save it in a file named `quotes_spider.py` under the `tutorial/spiders` directory in your project:

```
import scrapy

class QuotesSpider(scrapy.Spider):
    name = "quotes"

    def start_requests(self):
        urls = [
            'http://quotes.toscrape.com/page/1/',
            'http://quotes.toscrape.com/page/2/',
        ]
        for url in urls:
            yield scrapy.Request(url=url, callback=self.parse)

    def parse(self, response):
        page = response.url.split("/")[-2]
        filename = 'quotes-%s.html' % page
        with open(filename, 'wb') as f:
            f.write(response.body)
        self.log('Saved file %s' % filename)
```

As you can see, our Spider subclasses `scrapy.Spider` and defines some attributes and methods:

- `name`: identifies the Spider. It must be unique within a project, that is, you can't set the same name for different Spiders.
- `start_requests()`: must return an iterable of Requests (you can return a list of requests or write a generator function) which the Spider will begin to crawl from. Subsequent requests will be generated successively from these initial requests.
- `parse()`: a method that will be called to handle the response downloaded for each of the requests made. The response parameter is an instance of `TextResponse` that holds the page content and has further helpful methods to handle it.

The `parse()` method usually parses the response, extracting the scraped data as dicts and also finding new URLs to follow and creating new requests (`Request`) from them.

### How to run our spider

To put our spider to work, go to the project's top level directory and run:

```
scrapy crawl quotes
```



This command runs the spider with name `quotes` that we've just added, that will send some requests for the `quotes.toscrape.com` domain. You will get an output similar to this:

```
... (omitted for brevity)
2016-12-16 21:24:05 [scrapy.core.engine] INFO: Spider opened
2016-12-16 21:24:05 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pages/
↳min), scraped 0 items (at 0 items/min)
2016-12-16 21:24:05 [scrapy.extensions.telnet] DEBUG: Telnet console listening on 127.
↳0.0.1:6023
2016-12-16 21:24:05 [scrapy.core.engine] DEBUG: Crawled (404) <GET http://quotes.
↳toscraper.com/robots.txt> (referer: None)
2016-12-16 21:24:05 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://quotes.
↳toscraper.com/page/1/> (referer: None)
2016-12-16 21:24:05 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://quotes.
↳toscraper.com/page/2/> (referer: None)
2016-12-16 21:24:05 [quotes] DEBUG: Saved file quotes-1.html
2016-12-16 21:24:05 [quotes] DEBUG: Saved file quotes-2.html
2016-12-16 21:24:05 [scrapy.core.engine] INFO: Closing spider (finished)
...
```

Now, check the files in the current directory. You should notice that two new files have been created: *quotes-1.html* and *quotes-2.html*, with the content for the respective URLs, as our parse method instructs.

---

**Note:** If you are wondering why we haven't parsed the HTML yet, hold on, we will cover that soon.

---

## What just happened under the hood?

Scrapy schedules the `scrapy.Request` objects returned by the `start_requests` method of the Spider. Upon receiving a response for each one, it instantiates `Response` objects and calls the callback method associated with the request (in this case, the `parse` method) passing the response as argument.

## A shortcut to the `start_requests` method

Instead of implementing a `start_requests()` method that generates `scrapy.Request` objects from URLs, you can just define a `start_urls` class attribute with a list of URLs. This list will then be used by the default implementation of `start_requests()` to create the initial requests for your spider:

```
import scrapy

class QuotesSpider(scrapy.Spider):
    name = "quotes"
    start_urls = [
        'http://quotes.toscrape.com/page/1/',
        'http://quotes.toscrape.com/page/2/',
    ]

    def parse(self, response):
        page = response.url.split("/")[-2]
        filename = 'quotes-%s.html' % page
        with open(filename, 'wb') as f:
            f.write(response.body)
```

The `parse()` method will be called to handle each of the requests for those URLs, even though we haven't explicitly told Scrapy to do so. This happens because `parse()` is Scrapy's default callback method, which is called for requests without an explicitly assigned callback.

## Extracting data

The best way to learn how to extract data with Scrapy is trying selectors using the shell *Scrapy shell*. Run:

```
scrapy shell 'http://quotes.toscrape.com/page/1/'
```

**Note:** Remember to always enclose urls in quotes when running Scrapy shell from command-line, otherwise urls containing arguments (ie. `&` character) will not work.

On Windows, use double quotes instead:

```
scrapy shell "http://quotes.toscrape.com/page/1/"
```

You will see something like:

```
[ ... Scrapy log here ... ]
2016-09-19 12:09:27 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://quotes.
↪toscrape.com/page/1/> (referer: None)
[s] Available Scrapy objects:
[s]   scrapy       scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s]   crawler      <scrapy.crawler.Crawler object at 0x7fa91d888c90>
[s]   item         {}
[s]   request      <GET http://quotes.toscrape.com/page/1/>
[s]   response     <200 http://quotes.toscrape.com/page/1/>
[s]   settings     <scrapy.settings.Settings object at 0x7fa91d888c10>
[s]   spider       <DefaultSpider 'default' at 0x7fa91c8af990>
[s] Useful shortcuts:
[s]   shelp()       Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local objects
[s]   view(response) View response in a browser
>>>
```

Using the shell, you can try selecting elements using **CSS** with the response object:

```
>>> response.css('title')
[<Selector xpath='descendant-or-self::title' data='<title>Quotes to Scrape</title>'>]
```

The result of running `response.css('title')` is a list-like object called *SelectorList*, which represents a list of *Selector* objects that wrap around XML/HTML elements and allow you to run further queries to fine-grain the selection or extract the data.

To extract the text from the title above, you can do:

```
>>> response.css('title::text').extract()
['Quotes to Scrape']
```

There are two things to note here: one is that we've added `::text` to the CSS query, to mean we want to select only the text elements directly inside `<title>` element. If we don't specify `::text`, we'd get the full title element, including its tags:

```
>>> response.css('title').extract()
['<title>Quotes to Scrape</title>']
```

The other thing is that the result of calling `.extract()` is a list, because we're dealing with an instance of *SelectorList*. When you know you just want the first result, as in this case, you can do:

```
>>> response.css('title::text').extract_first()
'Quotes to Scrape'
```

As an alternative, you could've written:

```
>>> response.css('title::text')[0].extract()
'Quotes to Scrape'
```

However, using `.extract_first()` avoids an `IndexError` and returns `None` when it doesn't find any element matching the selection.

There's a lesson here: for most scraping code, you want it to be resilient to errors due to things not being found on a page, so that even if some parts fail to be scraped, you can at least get **some** data.

Besides the `extract()` and `extract_first()` methods, you can also use the `re()` method to extract using *regular expressions*:

```
>>> response.css('title::text').re(r'Quotes.*')
['Quotes to Scrape']
>>> response.css('title::text').re(r'Q\w+')
['Quotes']
>>> response.css('title::text').re(r'(\w+) to (\w+)')
['Quotes', 'Scrape']
```

In order to find the proper CSS selectors to use, you might find useful opening the response page from the shell in your web browser using `view(response)`. You can use your browser developer tools (see section about *Using your browser's Developer Tools for scraping*).

*Selector Gadget* is also a nice tool to quickly find CSS selector for visually selected elements, which works in many browsers.

## XPath: a brief intro

Besides *CSS*, Scrapy selectors also support using *XPath* expressions:

```
>>> response.xpath('//title')
[<Selector xpath='//title' data='<title>Quotes to Scrape</title>'>]
>>> response.xpath('//title/text()').extract_first()
'Quotes to Scrape'
```

XPath expressions are very powerful, and are the foundation of Scrapy Selectors. In fact, CSS selectors are converted to XPath under-the-hood. You can see that if you read closely the text representation of the selector objects in the shell.

While perhaps not as popular as CSS selectors, XPath expressions offer more power because besides navigating the structure, it can also look at the content. Using XPath, you're able to select things like: *select the link that contains the text "Next Page"*. This makes XPath very fitting to the task of scraping, and we encourage you to learn XPath even if you already know how to construct CSS selectors, it will make scraping much easier.

We won't cover much of XPath here, but you can read more about *using XPath with Scrapy Selectors here*. To learn more about XPath, we recommend [this tutorial to learn XPath through examples](#), and [this tutorial to learn "how to think in XPath"](#).

## Extracting quotes and authors

Now that you know a bit about selection and extraction, let's complete our spider by writing the code to extract the quotes from the web page.

Each quote in <http://quotes.toscrape.com> is represented by HTML elements that look like this:

```
<div class="quote">
  <span class="text">"The world as we have created it is a process of our
  thinking. It cannot be changed without changing our thinking."</span>
  <span>
    by <small class="author">Albert Einstein</small>
    <a href="/author/Albert-Einstein">(about)</a>
  </span>
  <div class="tags">
    Tags:
    <a class="tag" href="/tag/change/page/1/">change</a>
    <a class="tag" href="/tag/deep-thoughts/page/1/">deep-thoughts</a>
    <a class="tag" href="/tag/thinking/page/1/">thinking</a>
    <a class="tag" href="/tag/world/page/1/">world</a>
  </div>
</div>
```

Let's open up scrapy shell and play a bit to find out how to extract the data we want:

```
$ scrapy shell 'http://quotes.toscrape.com'
```

We get a list of selectors for the quote HTML elements with:

```
>>> response.css("div.quote")
```

Each of the selectors returned by the query above allows us to run further queries over their sub-elements. Let's assign the first selector to a variable, so that we can run our CSS selectors directly on a particular quote:

```
>>> quote = response.css("div.quote")[0]
```

Now, let's extract title, author and the tags from that quote using the quote object we just created:

```
>>> title = quote.css("span.text::text").extract_first()
>>> title
'"The world as we have created it is a process of our thinking. It cannot be changed_
↳without changing our thinking."'
>>> author = quote.css("small.author::text").extract_first()
>>> author
'Albert Einstein'
```

Given that the tags are a list of strings, we can use the `.extract()` method to get all of them:

```
>>> tags = quote.css("div.tags a.tag::text").extract()
>>> tags
['change', 'deep-thoughts', 'thinking', 'world']
```

Having figured out how to extract each bit, we can now iterate over all the quotes elements and put them together into a Python dictionary:

```
>>> for quote in response.css("div.quote"):
...     text = quote.css("span.text::text").extract_first()
...     author = quote.css("small.author::text").extract_first()
...     tags = quote.css("div.tags a.tag::text").extract()
...     print(dict(text=text, author=author, tags=tags))
{'tags': ['change', 'deep-thoughts', 'thinking', 'world'], 'author': 'Albert Einstein', 'text': '"The world as we have created it is a process of our thinking. It cannot be changed without changing our thinking."'}
{'tags': ['abilities', 'choices'], 'author': 'J.K. Rowling', 'text': '"It is our choices, Harry, that show what we truly are, far more than our abilities."'}
... a few more of these, omitted for brevity
>>>
```

### Extracting data in our spider

Let's get back to our spider. Until now, it doesn't extract any data in particular, just saves the whole HTML page to a local file. Let's integrate the extraction logic above into our spider.

A Scrapy spider typically generates many dictionaries containing the data extracted from the page. To do that, we use the `yield` Python keyword in the callback, as you can see below:

```
import scrapy

class QuotesSpider(scrapy.Spider):
    name = "quotes"
    start_urls = [
        'http://quotes.toscrape.com/page/1/',
        'http://quotes.toscrape.com/page/2/',
    ]

    def parse(self, response):
        for quote in response.css('div.quote'):
            yield {
                'text': quote.css('span.text::text').extract_first(),
                'author': quote.css('small.author::text').extract_first(),
                'tags': quote.css('div.tags a.tag::text').extract(),
            }
```

If you run this spider, it will output the extracted data with the log:

```
2016-09-19 18:57:19 [scrapy.core.scrapers] DEBUG: Scraped from <200 http://quotes.toscrape.com/page/1/>
{'tags': ['life', 'love'], 'author': 'André Gide', 'text': '"It is better to be hated for what you are than to be loved for what you are not."'}
2016-09-19 18:57:19 [scrapy.core.scrapers] DEBUG: Scraped from <200 http://quotes.toscrape.com/page/1/>
{'tags': ['edison', 'failure', 'inspirational', 'paraphrased'], 'author': 'Thomas A. Edison', 'text': '"I have not failed. I've just found 10,000 ways that won't work."'}
{'tags': [], 'author': '...', 'text': '...'}
```

### 2.3.3 Storing the scraped data

The simplest way to store the scraped data is by using *Feed exports*, with the following command:

```
scrapy crawl quotes -o quotes.json
```

That will generate an `quotes.json` file containing all scraped items, serialized in *JSON*.

For historic reasons, Scrapy appends to a given file instead of overwriting its contents. If you run this command twice without removing the file before the second time, you'll end up with a broken JSON file.

You can also use other formats, like *JSON Lines*:

```
scrapy crawl quotes -o quotes.jl
```

The *JSON Lines* format is useful because it's stream-like, you can easily append new records to it. It doesn't have the same problem of JSON when you run twice. Also, as each record is a separate line, you can process big files without having to fit everything in memory, there are tools like *JQ* to help doing that at the command-line.

In small projects (like the one in this tutorial), that should be enough. However, if you want to perform more complex things with the scraped items, you can write an *Item Pipeline*. A placeholder file for Item Pipelines has been set up for you when the project is created, in `tutorial/pipelines.py`. Though you don't need to implement any item pipelines if you just want to store the scraped items.

### 2.3.4 Following links

Let's say, instead of just scraping the stuff from the first two pages from <http://quotes.toscrape.com>, you want quotes from all the pages in the website.

Now that you know how to extract data from pages, let's see how to follow links from them.

First thing is to extract the link to the page we want to follow. Examining our page, we can see there is a link to the next page with the following markup:

```
<ul class="pager">
  <li class="next">
    <a href="/page/2/">Next <span aria-hidden="true">&rarr;</span></a>
  </li>
</ul>
```

We can try extracting it in the shell:

```
>>> response.css('li.next a').extract_first()
'<a href="/page/2/">Next <span aria-hidden="true">→</span></a>'
```

This gets the anchor element, but we want the attribute `href`. For that, Scrapy supports a CSS extension that let's you select the attribute contents, like this:

```
>>> response.css('li.next a::attr(href)').extract_first()
'/page/2/'
```

Let's see now our spider modified to recursively follow the link to the next page, extracting data from it:

```
import scrapy

class QuotesSpider(scrapy.Spider):
```

(continues on next page)

(continued from previous page)

```

name = "quotes"
start_urls = [
    'http://quotes.toscrape.com/page/1/',
]

def parse(self, response):
    for quote in response.css('div.quote'):
        yield {
            'text': quote.css('span.text::text').extract_first(),
            'author': quote.css('small.author::text').extract_first(),
            'tags': quote.css('div.tags a.tag::text').extract(),
        }

    next_page = response.css('li.next a::attr(href)').extract_first()
    if next_page is not None:
        next_page = response.urljoin(next_page)
        yield scrapy.Request(next_page, callback=self.parse)

```

Now, after extracting the data, the `parse()` method looks for the link to the next page, builds a full absolute URL using the `urljoin()` method (since the links can be relative) and yields a new request to the next page, registering itself as callback to handle the data extraction for the next page and to keep the crawling going through all the pages.

What you see here is Scrapy's mechanism of following links: when you yield a Request in a callback method, Scrapy will schedule that request to be sent and register a callback method to be executed when that request finishes.

Using this, you can build complex crawlers that follow links according to rules you define, and extract different kinds of data depending on the page it's visiting.

In our example, it creates a sort of loop, following all the links to the next page until it doesn't find one – handy for crawling blogs, forums and other sites with pagination.

## A shortcut for creating Requests

As a shortcut for creating Request objects you can use `response.follow`:

```

import scrapy

class QuotesSpider(scrapy.Spider):
    name = "quotes"
    start_urls = [
        'http://quotes.toscrape.com/page/1/',
    ]

    def parse(self, response):
        for quote in response.css('div.quote'):
            yield {
                'text': quote.css('span.text::text').extract_first(),
                'author': quote.css('span small::text').extract_first(),
                'tags': quote.css('div.tags a.tag::text').extract(),
            }

        next_page = response.css('li.next a::attr(href)').extract_first()
        if next_page is not None:
            yield response.follow(next_page, callback=self.parse)

```

Unlike `scrapy.Request`, `response.follow` supports relative URLs directly - no need to call `urljoin`. Note that `response.follow` just returns a `Request` instance; you still have to yield this `Request`.

You can also pass a selector to `response.follow` instead of a string; this selector should extract necessary attributes:

```
for href in response.css('li.next a::attr(href)'):
    yield response.follow(href, callback=self.parse)
```

For `<a>` elements there is a shortcut: `response.follow` uses their `href` attribute automatically. So the code can be shortened further:

```
for a in response.css('li.next a'):
    yield response.follow(a, callback=self.parse)
```

---

**Note:** `response.follow(response.css('li.next a'))` is not valid because `response.css` returns a list-like object with selectors for all results, not a single selector. A `for` loop like in the example above, or `response.follow(response.css('li.next a')[0])` is fine.

---

## More examples and patterns

Here is another spider that illustrates callbacks and following links, this time for scraping author information:

```
import scrapy

class AuthorSpider(scrapy.Spider):
    name = 'author'

    start_urls = ['http://quotes.toscrape.com/']

    def parse(self, response):
        # follow links to author pages
        for href in response.css('.author + a::attr(href)'):
            yield response.follow(href, self.parse_author)

        # follow pagination links
        for href in response.css('li.next a::attr(href)'):
            yield response.follow(href, self.parse)

    def parse_author(self, response):
        def extract_with_css(query):
            return response.css(query).extract_first().strip()

        yield {
            'name': extract_with_css('h3.author-title::text'),
            'birthdate': extract_with_css('.author-born-date::text'),
            'bio': extract_with_css('.author-description::text'),
        }
```

This spider will start from the main page, it will follow all the links to the authors pages calling the `parse_author` callback for each of them, and also the pagination links with the `parse` callback as we saw before.

Here we're passing callbacks to `response.follow` as positional arguments to make the code shorter; it also works for `scrapy.Request`.



The `parse_author` callback defines a helper function to extract and cleanup the data from a CSS query and yields the Python dict with the author data.

Another interesting thing this spider demonstrates is that, even if there are many quotes from the same author, we don't need to worry about visiting the same author page multiple times. By default, Scrapy filters out duplicated requests to URLs already visited, avoiding the problem of hitting servers too much because of a programming mistake. This can be configured by the setting `:setting:'DUPEFILTER_CLASS'`.

Hopefully by now you have a good understanding of how to use the mechanism of following links and callbacks with Scrapy.

As yet another example spider that leverages the mechanism of following links, check out the `CrawlSpider` class for a generic spider that implements a small rules engine that you can use to write your crawlers on top of it.

Also, a common pattern is to build an item with data from more than one page, using a *trick to pass additional data to the callbacks*.

## 2.3.5 Using spider arguments

You can provide command line arguments to your spiders by using the `-a` option when running them:

```
scrapy crawl quotes -o quotes-humor.json -a tag=humor
```

These arguments are passed to the Spider's `__init__` method and become spider attributes by default.

In this example, the value provided for the `tag` argument will be available via `self.tag`. You can use this to make your spider fetch only quotes with a specific tag, building the URL based on the argument:

```
import scrapy

class QuotesSpider(scrapy.Spider):
    name = "quotes"

    def start_requests(self):
        url = 'http://quotes.toscrape.com/'
        tag = getattr(self, 'tag', None)
        if tag is not None:
            url = url + 'tag/' + tag
        yield scrapy.Request(url, self.parse)

    def parse(self, response):
        for quote in response.css('div.quote'):
            yield {
                'text': quote.css('span.text::text').extract_first(),
                'author': quote.css('small.author::text').extract_first(),
            }

        next_page = response.css('li.next a::attr(href)').extract_first()
        if next_page is not None:
            yield response.follow(next_page, self.parse)
```

If you pass the `tag=humor` argument to this spider, you'll notice that it will only visit URLs from the `humor` tag, such as `http://quotes.toscrape.com/tag/humor`.

You can *learn more about handling spider arguments here*.

### 2.3.6 Next steps

This tutorial covered only the basics of Scrapy, but there's a lot of other features not mentioned here. Check the *What else?* section in *Scrapy* chapter for a quick overview of the most important ones.

You can continue from the section *Basic concepts* to know more about the command-line tool, spiders, selectors and other things the tutorial hasn't covered like modeling the scraped data. If you prefer to play with an example project, check the *Examples* section.

## 2.4 Examples

The best way to learn is with examples, and Scrapy is no exception. For this reason, there is an example Scrapy project named *quotesbot*, that you can use to play and learn more about Scrapy. It contains two spiders for <http://quotes.toscrape.com>, one using CSS selectors and another one using XPath expressions.

The *quotesbot* project is available at: <https://github.com/scrapy/quotesbot>. You can find more information about it in the project's README.

If you're familiar with git, you can checkout the code. Otherwise you can download the project as a zip file by clicking [here](#).

**Scrapy** Understand what Scrapy is and how it can help you.

Get Scrapy installed on your computer.

**Scrapy** Write your first Scrapy project.

**Examples** Learn more by playing with a pre-made Scrapy project.

### 3.1 Command line tool

New in version 0.10.

Scrapy is controlled through the `scrapy` command-line tool, to be referred here as the “Scrapy tool” to differentiate it from the sub-commands, which we just call “commands” or “Scrapy commands”.

The Scrapy tool provides several commands, for multiple purposes, and each one accepts a different set of arguments and options.

(The `scrapy deploy` command has been removed in 1.0 in favor of the standalone `scrapyd-deploy`. See [Deploying your project](#).)

#### 3.1.1 Configuration settings

Scrapy will look for configuration parameters in ini-style `scrapy.cfg` files in standard locations:

1. `/etc/scrapy.cfg` or `c:\scrapy\scrapy.cfg` (system-wide),
2. `~/.config/scrapy.cfg` (`$XDG_CONFIG_HOME`) and `~/.scrapy.cfg` (`$HOME`) for global (user-wide) settings, and
3. `scrapy.cfg` inside a scrapy project’s root (see next section).

Settings from these files are merged in the listed order of preference: user-defined values have higher priority than system-wide defaults and project-wide settings will override all others, when defined.

Scrapy also understands, and can be configured through, a number of environment variables. Currently these are:

- `SCRAPY_SETTINGS_MODULE` (see [Designating the settings](#))
- `SCRAPY_PROJECT`
- `SCRAPY_PYTHON_SHELL` (see [Scrapy shell](#))

### 3.1.2 Default structure of Scrapy projects

Before delving into the command-line tool and its sub-commands, let's first understand the directory structure of a Scrapy project.

Though it can be modified, all Scrapy projects have the same file structure by default, similar to this:

```
scrapy.cfg
myproject/
  __init__.py
  items.py
  middlewares.py
  pipelines.py
  settings.py
  spiders/
    __init__.py
    spider1.py
    spider2.py
    ...
```

The directory where the `scrapy.cfg` file resides is known as the *project root directory*. That file contains the name of the python module that defines the project settings. Here is an example:

```
[settings]
default = myproject.settings
```

### 3.1.3 Using the scrapy tool

You can start by running the Scrapy tool with no arguments and it will print some usage help and the available commands:

```
Scrapy X.Y - no active project

Usage:
  scrapy <command> [options] [args]

Available commands:
  crawl          Run a spider
  fetch          Fetch a URL using the Scrapy downloader
  [...]
```

The first line will print the currently active project if you're inside a Scrapy project. In this example it was run from outside a project. If run from inside a project it would have printed something like this:

```
Scrapy X.Y - project: myproject

Usage:
  scrapy <command> [options] [args]

[...]
```

### Creating projects

The first thing you typically do with the `scrapy` tool is create your Scrapy project:

```
scrapy startproject myproject [project_dir]
```

That will create a Scrapy project under the `project_dir` directory. If `project_dir` wasn't specified, `project_dir` will be the same as `myproject`.

Next, you go inside the new project directory:

```
cd project_dir
```

And you're ready to use the `scrapy` command to manage and control your project from there.

## Controlling projects

You use the `scrapy` tool from inside your projects to control and manage them.

For example, to create a new spider:

```
scrapy genspider mydomain mydomain.com
```

Some Scrapy commands (like **`crawl`**) must be run from inside a Scrapy project. See the [commands reference](#) below for more information on which commands must be run from inside projects, and which not.

Also keep in mind that some commands may have slightly different behaviours when running them from inside projects. For example, the `fetch` command will use spider-overridden behaviours (such as the `user_agent` attribute to override the user-agent) if the url being fetched is associated with some specific spider. This is intentional, as the `fetch` command is meant to be used to check how spiders are downloading pages.

### 3.1.4 Available tool commands

This section contains a list of the available built-in commands with a description and some usage examples. Remember, you can always get more info about each command by running:

```
scrapy <command> -h
```

And you can see all available commands with:

```
scrapy -h
```

There are two kinds of commands, those that only work from inside a Scrapy project (Project-specific commands) and those that also work without an active Scrapy project (Global commands), though they may behave slightly different when running from inside a project (as they would use the project overridden settings).

Global commands:

- **`startproject`**
- **`genspider`**
- **`settings`**
- **`runspider`**
- **`shell`**
- **`fetch`**
- **`view`**
- **`version`**

Project-only commands:

- **crawl**
- **check**
- **list**
- **edit**
- **parse**
- **bench**

### startproject

- Syntax: `scrapy startproject <project_name> [project_dir]`
- Requires project: *no*

Creates a new Scrapy project named `project_name`, under the `project_dir` directory. If `project_dir` wasn't specified, `project_dir` will be the same as `project_name`.

Usage example:

```
$ scrapy startproject myproject
```

### genspider

- Syntax: `scrapy genspider [-t template] <name> <domain>`
- Requires project: *no*

Create a new spider in the current folder or in the current project's `spiders` folder, if called from inside a project. The `<name>` parameter is set as the spider's name, while `<domain>` is used to generate the `allowed_domains` and `start_urls` spider's attributes.

Usage example:

```
$ scrapy genspider -l
Available templates:
  basic
  crawl
  csvfeed
  xmlfeed

$ scrapy genspider example example.com
Created spider 'example' using template 'basic'

$ scrapy genspider -t crawl scrapyorg scrapy.org
Created spider 'scrapyorg' using template 'crawl'
```

This is just a convenience shortcut command for creating spiders based on pre-defined templates, but certainly not the only way to create spiders. You can just create the spider source code files yourself, instead of using this command.

### crawl

- Syntax: `scrapy crawl <spider>`

- Requires project: *yes*

Start crawling using a spider.

Usage examples:

```
$ scrapy crawl myspider
[ ... myspider starts crawling ... ]
```

## check

- Syntax: `scrapy check [-l] <spider>`
- Requires project: *yes*

Run contract checks.

Usage examples:

```
$ scrapy check -l
first_spider
  * parse
  * parse_item
second_spider
  * parse
  * parse_item

$ scrapy check
[FAILED] first_spider:parse_item
>>> 'RetailPricex' field is missing

[FAILED] first_spider:parse
>>> Returned 92 requests, expected 0..4
```

## list

- Syntax: `scrapy list`
- Requires project: *yes*

List all available spiders in the current project. The output is one spider per line.

Usage example:

```
$ scrapy list
spider1
spider2
```

## edit

- Syntax: `scrapy edit <spider>`
- Requires project: *yes*

Edit the given spider using the editor defined in the `EDITOR` environment variable or (if unset) the **:setting:‘EDITOR‘** setting.

This command is provided only as a convenience shortcut for the most common case, the developer is of course free to choose any tool or IDE to write and debug spiders.

Usage example:

```
$ scrapy edit spider1
```

### fetch

- Syntax: `scrapy fetch <url>`
- Requires project: *no*

Downloads the given URL using the Scrapy downloader and writes the contents to standard output.

The interesting thing about this command is that it fetches the page how the spider would download it. For example, if the spider has a `USER_AGENT` attribute which overrides the User Agent, it will use that one.

So this command can be used to “see” how your spider would fetch a certain page.

If used outside a project, no particular per-spider behaviour would be applied and it will just use the default Scrapy downloader settings.

Supported options:

- `--spider=SPIDER`: bypass spider autodetection and force use of specific spider
- `--headers`: print the response’s HTTP headers instead of the response’s body
- `--no-redirect`: do not follow HTTP 3xx redirects (default is to follow them)

Usage examples:

```
$ scrapy fetch --nolog http://www.example.com/some/page.html
[ ... html content here ... ]

$ scrapy fetch --nolog --headers http://www.example.com/
{'Accept-Ranges': ['bytes'],
 'Age': ['1263'],
 'Connection': ['close'],
 'Content-Length': ['596'],
 'Content-Type': ['text/html; charset=UTF-8'],
 'Date': ['Wed, 18 Aug 2010 23:59:46 GMT'],
 'Etag': ['"573c1-254-48c9c87349680"'],
 'Last-Modified': ['Fri, 30 Jul 2010 15:30:18 GMT'],
 'Server': ['Apache/2.2.3 (CentOS)']}
```

### view

- Syntax: `scrapy view <url>`
- Requires project: *no*

Opens the given URL in a browser, as your Scrapy spider would “see” it. Sometimes spiders see pages differently from regular users, so this can be used to check what the spider “sees” and confirm it’s what you expect.

Supported options:

- `--spider=SPIDER`: bypass spider autodetection and force use of specific spider
- `--no-redirect`: do not follow HTTP 3xx redirects (default is to follow them)



Usage example:

```
$ scrapy view http://www.example.com/some/page.html
[ ... browser starts ... ]
```

## shell

- Syntax: `scrapy shell [url]`
- Requires project: *no*

Starts the Scrapy shell for the given URL (if given) or empty if no URL is given. Also supports UNIX-style local file paths, either relative with `./` or `../` prefixes or absolute file paths. See [Scrapy shell](#) for more info.

Supported options:

- `--spider=SPIDER`: bypass spider autodetection and force use of specific spider
- `-c code`: evaluate the code in the shell, print the result and exit
- `--no-redirect`: do not follow HTTP 3xx redirects (default is to follow them); this only affects the URL you may pass as argument on the command line; once you are inside the shell, `fetch(url)` will still follow HTTP redirects by default.

Usage example:

```
$ scrapy shell http://www.example.com/some/page.html
[ ... scrapy shell starts ... ]

$ scrapy shell --nolog http://www.example.com/ -c '(response.status, response.url)'
(200, 'http://www.example.com/')

# shell follows HTTP redirects by default
$ scrapy shell --nolog http://httpbin.org/redirect-to?url=http%3A%2F%2Fexample.com%2F
↪ -c '(response.status, response.url)'
(200, 'http://example.com/')

# you can disable this with --no-redirect
# (only for the URL passed as command line argument)
$ scrapy shell --no-redirect --nolog http://httpbin.org/redirect-to?url=http%3A%2F
↪ %2Fexample.com%2F -c '(response.status, response.url)'
(302, 'http://httpbin.org/redirect-to?url=http%3A%2F%2Fexample.com%2F')
```

## parse

- Syntax: `scrapy parse <url> [options]`
- Requires project: *yes*

Fetches the given URL and parses it with the spider that handles it, using the method passed with the `--callback` option, or `parse` if not given.

Supported options:

- `--spider=SPIDER`: bypass spider autodetection and force use of specific spider
- `--a NAME=VALUE`: set spider argument (may be repeated)
- `--callback` or `-c`: spider method to use as callback for parsing the response

- `--meta` or `-m`: additional request meta that will be passed to the callback request. This must be a valid json string. Example: `-meta='{"foo": "bar"}'`
- `--pipelines`: process items through pipelines
- `--rules` or `-r`: use *CrawlSpider* rules to discover the callback (i.e. spider method) to use for parsing the response
- `--noitems`: don't show scraped items
- `--nolinks`: don't show extracted links
- `--nocolour`: avoid using pygments to colorize the output
- `--depth` or `-d`: depth level for which the requests should be followed recursively (default: 1)
- `--verbose` or `-v`: display information for each depth level

Usage example:

```
$ scrapy parse http://www.example.com/ -c parse_item
[ ... scrapy log lines crawling example.com spider ... ]

>>> STATUS DEPTH LEVEL 1 <<<
# Scraped Items -----
[{'name': u'Example item',
  'category': u'Furniture',
  'length': u'12 cm'}]

# Requests -----
[]
```

## settings

- Syntax: `scrapy settings [options]`
- Requires project: *no*

Get the value of a Scrapy setting.

If used inside a project it'll show the project setting value, otherwise it'll show the default Scrapy value for that setting.

Example usage:

```
$ scrapy settings --get BOT_NAME
scrapybot
$ scrapy settings --get DOWNLOAD_DELAY
0
```

## runspider

- Syntax: `scrapy runspider <spider_file.py>`
- Requires project: *no*

Run a spider self-contained in a Python file, without having to create a project.

Example usage:

```
$ scrapy runspider myspider.py
[ ... spider starts crawling ... ]
```

## version

- Syntax: `scrapy version [-v]`
- Requires project: *no*

Prints the Scrapy version. If used with `-v` it also prints Python, Twisted and Platform info, which is useful for bug reports.

## bench

New in version 0.17.

- Syntax: `scrapy bench`
- Requires project: *no*

Run a quick benchmark test. *Benchmarking*.

## 3.1.5 Custom project commands

You can also add your custom project commands by using the **:setting:‘COMMANDS\_MODULE’** setting. See the Scrapy commands in [scrapy/commands](#) for examples on how to implement your commands.

### COMMANDS\_MODULE

Default: `''` (empty string)

A module to use for looking up custom Scrapy commands. This is used to add custom commands for your Scrapy project.

Example:

```
COMMANDS_MODULE = 'mybot.commands'
```

### Register commands via setup.py entry points

---

**Note:** This is an experimental feature, use with caution.

---

You can also add Scrapy commands from an external library by adding a `scrapy.commands` section in the entry points of the library `setup.py` file.

The following example adds `my_command` command:

```
from setuptools import setup, find_packages

setup(name='scrapy-mymodule',
      entry_points={
          'scrapy.commands': [
              'my_command=my_scrapy_module.commands:MyCommand',
          ],
      },
)
```

## 3.2 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.

### 3.2.1 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 *Request* 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 initializing 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)]
```

(continues on next page)

(continued from previous page)

```
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 **:signal:'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():
```

(continues on next page)

(continued from previous page)

```

        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)

```

### 3.2.2 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'

```

(continues on next page)

(continued from previous page)

```
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 Scrapy `schedule.json` API. See [Scrapy documentation](#).

### 3.2.3 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+)')
```

(continues on next page)

(continued from previous page)

```

        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 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']
```

(continues on next page)

(continued from previous page)

```
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 ...
```

## 3.3 Selectors

When you're scraping web pages, the most common task you need to perform is to extract data from the HTML source. There are several libraries available to achieve this:

- [BeautifulSoup](#) is a very popular web scraping library among Python programmers which constructs a Python object based on the structure of the HTML code and also deals with bad markup reasonably well, but it has one drawback: it's slow.
- [lxml](#) is an XML parsing library (which also parses HTML) with a pythonic API based on [ElementTree](#). ([lxml](#) is not part of the Python standard library.)

Scrapy comes with its own mechanism for extracting data. They're called selectors because they "select" certain parts of the HTML document specified either by [XPath](#) or [CSS](#) expressions.

[XPath](#) is a language for selecting nodes in XML documents, which can also be used with HTML. [CSS](#) is a language for applying styles to HTML documents. It defines selectors to associate those styles with specific HTML elements.

Scrapy selectors are built over the [lxml](#) library, which means they're very similar in speed and parsing accuracy.

This page explains how selectors work and describes their API which is very small and simple, unlike the [lxml](#) API which is much bigger because the [lxml](#) library can be used for many other tasks, besides selecting markup documents.

For a complete reference of the selectors API see [Selector reference](#)

### 3.3.1 Using selectors

#### Constructing selectors

Scrapy selectors are instances of *Selector* class constructed by passing **text** or *TextResponse* object. It automatically chooses the best parsing rules (XML vs HTML) based on input type:

```
>>> from scrapy.selector import Selector
>>> from scrapy.http import HtmlResponse
```

Constructing from text:

```
>>> body = '<html><body><span>good</span></body></html>'
>>> Selector(text=body).xpath('//span/text()').extract()
[u'good']
```

Constructing from response:

```
>>> response = HtmlResponse(url='http://example.com', body=body)
>>> Selector(response=response).xpath('//span/text()').extract()
[u'good']
```

For convenience, response objects expose a selector on *.selector* attribute, it's totally OK to use this shortcut when possible:

```
>>> response.selector.xpath('//span/text()').extract()
[u'good']
```

#### Using selectors

To explain how to use the selectors we'll use the *Scrapy shell* (which provides interactive testing) and an example page located in the Scrapy documentation server:

[https://doc.scrapy.org/en/latest/\\_static/selectors-sample1.html](https://doc.scrapy.org/en/latest/_static/selectors-sample1.html)

Here's its HTML code:

First, let's open the shell:

```
scrapy shell https://doc.scrapy.org/en/latest/_static/selectors-sample1.html
```

Then, after the shell loads, you'll have the response available as *response* shell variable, and its attached selector in *response.selector* attribute.

Since we're dealing with HTML, the selector will automatically use an HTML parser.

So, by looking at the *HTML code* of that page, let's construct an XPath for selecting the text inside the title tag:

```
>>> response.selector.xpath('//title/text()')
[<Selector (text) xpath=//title/text()>]
```

Querying responses using XPath and CSS is so common that responses include two convenience shortcuts: *response.xpath()* and *response.css()*:

```
>>> response.xpath('//title/text()')
[<Selector (text) xpath=//title/text()>]
>>> response.css('title::text')
[<Selector (text) xpath=//title/text()>]
```

As you can see, `.xpath()` and `.css()` methods return a *SelectorList* instance, which is a list of new selectors. This API can be used for quickly selecting nested data:

```
>>> response.css('img').xpath('@src').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']
```

To actually extract the textual data, you must call the selector `.extract()` method, as follows:

```
>>> response.xpath('//title/text()').extract()
[u'Example website']
```

If you want to extract only first matched element, you can call the selector `.extract_first()`

```
>>> response.xpath('//div[@id="images"]/a/text()').extract_first()
u'Name: My image 1 '
```

It returns `None` if no element was found:

```
>>> response.xpath('//div[@id="not-exists"]/text()').extract_first() is None
True
```

A default return value can be provided as an argument, to be used instead of `None`:

```
>>> response.xpath('//div[@id="not-exists"]/text()').extract_first(default='not-found')
'not-found'
```

Notice that CSS selectors can select text or attribute nodes using CSS3 pseudo-elements:

```
>>> response.css('title::text').extract()
[u'Example website']
```

Now we're going to get the base URL and some image links:

```
>>> response.xpath('//base/@href').extract()
[u'http://example.com/']

>>> response.css('base::attr(href)').extract()
[u'http://example.com/']

>>> response.xpath('//a[contains(@href, "image")]/@href').extract()
[u'image1.html',
 u'image2.html',
 u'image3.html',
 u'image4.html',
 u'image5.html']

>>> response.css('a[href*=image]::attr(href)').extract()
[u'image1.html',
 u'image2.html',
 u'image3.html',
 u'image4.html',
 u'image5.html']
```

(continues on next page)



(continued from previous page)

```
>>> response.xpath('//a[contains(@href, "image")]/img/@src').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']

>>> response.css('a[href*=image] img::attr(src)').extract()
[u'image1_thumb.jpg',
 u'image2_thumb.jpg',
 u'image3_thumb.jpg',
 u'image4_thumb.jpg',
 u'image5_thumb.jpg']
```

## Nesting selectors

The selection methods (`.xpath()` or `.css()`) return a list of selectors of the same type, so you can call the selection methods for those selectors too. Here's an example:

```
>>> links = response.xpath('//a[contains(@href, "image")]')
>>> links.extract()
[u'<a href="image1.html">Name: My image 1 <br></a>',
 u'<a href="image2.html">Name: My image 2 <br></a>',
 u'<a href="image3.html">Name: My image 3 <br></a>',
 u'<a href="image4.html">Name: My image 4 <br></a>',
 u'<a href="image5.html">Name: My image 5 <br></a>']

>>> for index, link in enumerate(links):
...     args = (index, link.xpath('@href').extract(), link.xpath('img/@src').
...     ↪extract())
...     print 'Link number %d points to url %s and image %s' % args

Link number 0 points to url [u'image1.html'] and image [u'image1_thumb.jpg']
Link number 1 points to url [u'image2.html'] and image [u'image2_thumb.jpg']
Link number 2 points to url [u'image3.html'] and image [u'image3_thumb.jpg']
Link number 3 points to url [u'image4.html'] and image [u'image4_thumb.jpg']
Link number 4 points to url [u'image5.html'] and image [u'image5_thumb.jpg']
```

## Using selectors with regular expressions

*Selector* also has a `.re()` method for extracting data using regular expressions. However, unlike using `.xpath()` or `.css()` methods, `.re()` returns a list of unicode strings. So you can't construct nested `.re()` calls.

Here's an example used to extract image names from the *HTML code* above:

```
>>> response.xpath('//a[contains(@href, "image")]/text()').re(r'Name:\s*(.*)')
[u'My image 1',
 u'My image 2',
 u'My image 3',
 u'My image 4',
 u'My image 5']
```

There's an additional helper reciprocating `.extract_first()` for `.re()`, named `.re_first()`. Use it to extract just the first matching string:

```
>>> response.xpath('//a[contains(@href, "image")]/text()').re_first(r'Name:\s*(.*)')
u'My image 1'
```

## Working with relative XPath

Keep in mind that if you are nesting selectors and use an XPath that starts with `/`, that XPath will be absolute to the document and not relative to the Selector you're calling it from.

For example, suppose you want to extract all `<p>` elements inside `<div>` elements. First, you would get all `<div>` elements:

```
>>> divs = response.xpath('//div')
```

At first, you may be tempted to use the following approach, which is wrong, as it actually extracts all `<p>` elements from the document, not only those inside `<div>` elements:

```
>>> for p in divs.xpath('/p'): # this is wrong - gets all <p> from the whole_
↳ document
...     print p.extract()
```

This is the proper way to do it (note the dot prefixing the `./p` XPath):

```
>>> for p in divs.xpath('./p'): # extracts all <p> inside
...     print p.extract()
```

Another common case would be to extract all direct `<p>` children:

```
>>> for p in divs.xpath('p'):
...     print p.extract()
```

For more details about relative XPath see the [Location Paths](#) section in the XPath specification.

## Variables in XPath expressions

XPath allows you to reference variables in your XPath expressions, using the `$somevariable` syntax. This is somewhat similar to parameterized queries or prepared statements in the SQL world where you replace some arguments in your queries with placeholders like `?`, which are then substituted with values passed with the query.

Here's an example to match an element based on its "id" attribute value, without hard-coding it (that was shown previously):

```
>>> # ` $val ` used in the expression, a ` val ` argument needs to be passed
>>> response.xpath('//div[@id=$val]/a/text()', val='images').extract_first()
u'Name: My image 1'
```

Here's another example, to find the "id" attribute of a `<div>` tag containing five `<a>` children (here we pass the value 5 as an integer):

```
>>> response.xpath('//div[count(a)=$cnt]/@id', cnt=5).extract_first()
u'images'
```

All variable references must have a binding value when calling `.xpath()` (otherwise you'll get a `ValueError: XPath error: exception`). This is done by passing as many named arguments as necessary.

`parsel`, the library powering Scrapy selectors, has more details and examples on [XPath variables](#).

## Using EXSLT extensions

Being built atop `lxml`, Scrapy selectors also support some [EXSLT](#) extensions and come with these pre-registered namespaces to use in XPath expressions:

prefix	namespace	usage
re	<a href="http://exslt.org/regular-expressions">http://exslt.org/regular-expressions</a>	<a href="#">regular expressions</a>
set	<a href="http://exslt.org/sets">http://exslt.org/sets</a>	<a href="#">set manipulation</a>

## Regular expressions

The `test()` function, for example, can prove quite useful when XPath's `starts-with()` or `contains()` are not sufficient.

Example selecting links in list item with a “class” attribute ending with a digit:

```
>>> from scrapy import Selector
>>> doc = """
... <div>
...     <ul>
...         <li class="item-0"><a href="link1.html">first item</a></li>
...         <li class="item-1"><a href="link2.html">second item</a></li>
...         <li class="item-inactive"><a href="link3.html">third item</a></li>
...         <li class="item-1"><a href="link4.html">fourth item</a></li>
...         <li class="item-0"><a href="link5.html">fifth item</a></li>
...     </ul>
... </div>
... """
>>> sel = Selector(text=doc, type="html")
>>> sel.xpath('//li//@href').extract()
[u'link1.html', u'link2.html', u'link3.html', u'link4.html', u'link5.html']
>>> sel.xpath('//li[re:test(@class, "item-\d$")]/@href').extract()
[u'link1.html', u'link2.html', u'link4.html', u'link5.html']
>>>
```

**Warning:** C library `libxslt` doesn't natively support EXSLT regular expressions so `lxml`'s implementation uses hooks to Python's `re` module. Thus, using regexp functions in your XPath expressions may add a small performance penalty.

## Set operations

These can be handy for excluding parts of a document tree before extracting text elements for example.

Example extracting microdata (sample content taken from <http://schema.org/Product>) with groups of `itemscope` and corresponding `itemprops`:

```
>>> doc = """
... <div itemscope itemtype="http://schema.org/Product">
...   <span itemprop="name">Kenmore White 17" Microwave</span>
...   
...   <div itemprop="aggregateRating"
...     itemscope itemtype="http://schema.org/AggregateRating">
...     Rated <span itemprop="ratingValue">3.5</span>/5
...     based on <span itemprop="reviewCount">11</span> customer reviews
...   </div>
...
...   <div itemprop="offers" itemscope itemtype="http://schema.org/Offer">
...     <span itemprop="price">$55.00</span>
...     <link itemprop="availability" href="http://schema.org/InStock" />In stock
...   </div>
...
...   Product description:
...   <span itemprop="description">0.7 cubic feet countertop microwave.
...   Has six preset cooking categories and convenience features like
...   Add-A-Minute and Child Lock.</span>
...
...   Customer reviews:
...
...   <div itemprop="review" itemscope itemtype="http://schema.org/Review">
...     <span itemprop="name">Not a happy camper</span> -
...     by <span itemprop="author">Ellie</span>,
...     <meta itemprop="datePublished" content="2011-04-01">April 1, 2011
...     <div itemprop="reviewRating" itemscope itemtype="http://schema.org/Rating">
...       <meta itemprop="worstRating" content = "1">
...       <span itemprop="ratingValue">1</span>/
...       <span itemprop="bestRating">5</span>stars
...     </div>
...     <span itemprop="description">The lamp burned out and now I have to replace
...     it. </span>
...   </div>
...
...   <div itemprop="review" itemscope itemtype="http://schema.org/Review">
...     <span itemprop="name">Value purchase</span> -
...     by <span itemprop="author">Lucas</span>,
...     <meta itemprop="datePublished" content="2011-03-25">March 25, 2011
...     <div itemprop="reviewRating" itemscope itemtype="http://schema.org/Rating">
...       <meta itemprop="worstRating" content = "1"/>
...       <span itemprop="ratingValue">4</span>/
...       <span itemprop="bestRating">5</span>stars
...     </div>
...     <span itemprop="description">Great microwave for the price. It is small and
...     fits in my apartment.</span>
...   </div>
...   ...
... </div>
... """
>>> sel = Selector(text=doc, type="html")
>>> for scope in sel.xpath('//div[@itemscope]'):
...     print "current scope:", scope.xpath('@itemtype').extract()
...     props = scope.xpath(''''
...         set: difference(./descendant::*/@itemprop,
...             .//*[[@itemscope]/*/@itemprop)'''
...     )
...     print "    properties:", props.extract()
```

(continues on next page)

(continued from previous page)

```

...     print

current scope: [u'http://schema.org/Product']
  properties: [u'name', u'aggregateRating', u'offers', u'description', u'review', u
↪ 'review']

current scope: [u'http://schema.org/AggregateRating']
  properties: [u'ratingValue', u'reviewCount']

current scope: [u'http://schema.org/Offer']
  properties: [u'price', u'availability']

current scope: [u'http://schema.org/Review']
  properties: [u'name', u'author', u'datePublished', u'reviewRating', u'description
↪ '']

current scope: [u'http://schema.org/Rating']
  properties: [u'worstRating', u'ratingValue', u'bestRating']

current scope: [u'http://schema.org/Review']
  properties: [u'name', u'author', u'datePublished', u'reviewRating', u'description
↪ '']

current scope: [u'http://schema.org/Rating']
  properties: [u'worstRating', u'ratingValue', u'bestRating']

>>>

```

Here we first iterate over `itemscope` elements, and for each one, we look for all `itemprops` elements and exclude those that are themselves inside another `itemscope`.

### Some XPath tips

Here are some tips that you may find useful when using XPath with Scrapy selectors, based on [this post from ScrapingHub's blog](#). If you are not much familiar with XPath yet, you may want to take a look first at [this XPath tutorial](#).

### Using text nodes in a condition

When you need to use the text content as argument to an [XPath string function](#), avoid using `./text()` and use just `.` instead.

This is because the expression `./text()` yields a collection of text elements – a *node-set*. And when a node-set is converted to a string, which happens when it is passed as argument to a string function like `contains()` or `starts-with()`, it results in the text for the first element only.

Example:

```

>>> from scrapy import Selector
>>> sel = Selector(text='<a href="#">Click here to go to the <strong>Next Page</
↪ strong></a>')

```

Converting a *node-set* to string:

```
>>> sel.xpath('//a/text()').extract() # take a peek at the node-set
[u'Click here to go to the ', u'Next Page']
>>> sel.xpath("string(//a[1]/text())").extract() # convert it to string
[u'Click here to go to the ']
```

A *node* converted to a string, however, puts together the text of itself plus of all its descendants:

```
>>> sel.xpath("//a[1]").extract() # select the first node
[u'<a href="#">Click here to go to the <strong>Next Page</strong></a>']
>>> sel.xpath("string(//a[1])").extract() # convert it to string
[u'Click here to go to the Next Page']
```

So, using the `./text()` node-set won't select anything in this case:

```
>>> sel.xpath("//a[contains(./text(), 'Next Page')]").extract()
[]
```

But using the `.` to mean the node, works:

```
>>> sel.xpath("//a[contains(., 'Next Page')]").extract()
[u'<a href="#">Click here to go to the <strong>Next Page</strong></a>']
```

## Beware of the difference between `//node[1]` and `(//node)[1]`

`//node[1]` selects all the nodes occurring first under their respective parents.

`(//node)[1]` selects all the nodes in the document, and then gets only the first of them.

Example:

```
>>> from scrapy import Selector
>>> sel = Selector(text="""
....: <ul class="list">
....:   <li>1</li>
....:   <li>2</li>
....:   <li>3</li>
....: </ul>
....: <ul class="list">
....:   <li>4</li>
....:   <li>5</li>
....:   <li>6</li>
....: </ul>""")
>>> xp = lambda x: sel.xpath(x).extract()
```

This gets all first `<li>` elements under whatever it is its parent:

```
>>> xp("//li[1]")
[u'<li>1</li>', u'<li>4</li>']
```

And this gets the first `<li>` element in the whole document:

```
>>> xp("(//li)[1]")
[u'<li>1</li>']
```

This gets all first `<li>` elements under an `<ul>` parent:

```
>>> xp("//ul/li[1]")
[u'<li>1</li>', u'<li>4</li>']
```

And this gets the first `<li>` element under an `<ul>` parent in the whole document:

```
>>> xp("//ul/li)[1]")
[u'<li>1</li>']
```

## When querying by class, consider using CSS

Because an element can contain multiple CSS classes, the XPath way to select elements by class is the rather verbose:

```
*[contains(concat(' ', normalize-space(@class), ' '), ' someclass ')]
```

If you use `@class='someclass'` you may end up missing elements that have other classes, and if you just use `contains(@class, 'someclass')` to make up for that you may end up with more elements that you want, if they have a different class name that shares the string `someclass`.

As it turns out, Scrapy selectors allow you to chain selectors, so most of the time you can just select by class using CSS and then switch to XPath when needed:

```
>>> from scrapy import Selector
>>> sel = Selector(text='<div class="hero shout"><time datetime="2014-07-23 19:00">
↳Special date</time></div>')
>>> sel.css('.shout').xpath('./time/@datetime').extract()
[u'2014-07-23 19:00']
```

This is cleaner than using the verbose XPath trick shown above. Just remember to use the `.` in the XPath expressions that will follow.

## 3.3.2 Built-in Selectors reference

### Selector objects

**class** scrapy.selector.Selector (response=None, text=None, type=None)

An instance of *Selector* is a wrapper over response to select certain parts of its content.

*response* is an *HtmlResponse* or an *XmlResponse* object that will be used for selecting and extracting data.

*text* is a unicode string or utf-8 encoded text for cases when a *response* isn't available. Using *text* and *response* together is undefined behavior.

*type* defines the selector type, it can be "html", "xml" or None (default).

If *type* is None, the selector automatically chooses the best type based on *response* type (see below), or defaults to "html" in case it is used together with *text*.

If *type* is None and a *response* is passed, the selector type is inferred from the response type as follows:

- "html" for *HtmlResponse* type
- "xml" for *XmlResponse* type
- "html" for anything else

Otherwise, if `type` is set, the selector type will be forced and no detection will occur.

**xpath** (*query*)

Find nodes matching the `xpath query` and return the result as a `SelectorList` instance with all elements flattened. List elements implement `Selector` interface too.

`query` is a string containing the XPATH query to apply.

---

**Note:** For convenience, this method can be called as `response.xpath()`

---

**css** (*query*)

Apply the given CSS selector and return a `SelectorList` instance.

`query` is a string containing the CSS selector to apply.

In the background, CSS queries are translated into XPath queries using `cssselect` library and run `.xpath()` method.

---

**Note:** For convenience this method can be called as `response.css()`

---

**extract** ()

Serialize and return the matched nodes as a list of unicode strings. Percent encoded content is unquoted.

**re** (*regex*)

Apply the given regex and return a list of unicode strings with the matches.

`regex` can be either a compiled regular expression or a string which will be compiled to a regular expression using `re.compile(regex)`

---

**Note:** Note that `re()` and `re_first()` both decode HTML entities (except `<`; and `&`).

---

**register\_namespace** (*prefix, uri*)

Register the given namespace to be used in this `Selector`. Without registering namespaces you can't select or extract data from non-standard namespaces. See examples below.

**remove\_namespaces** ()

Remove all namespaces, allowing to traverse the document using namespace-less xpaths. See example below.

**\_\_nonzero\_\_** ()

Returns `True` if there is any real content selected or `False` otherwise. In other words, the boolean value of a `Selector` is given by the contents it selects.

## SelectorList objects

**class** scrapy.selector.SelectorList

The `SelectorList` class is a subclass of the builtin `list` class, which provides a few additional methods.

**xpath** (*query*)

Call the `.xpath()` method for each element in this list and return their results flattened as another `SelectorList`.

`query` is the same argument as the one in `Selector.xpath()`



**css (query)**

Call the `.css()` method for each element in this list and return their results flattened as another *SelectorList*.

query is the same argument as the one in *Selector.css()*

**extract ()**

Call the `.extract()` method for each element in this list and return their results flattened, as a list of unicode strings.

**re ()**

Call the `.re()` method for each element in this list and return their results flattened, as a list of unicode strings.

## Selector examples on HTML response

Here's a couple of *Selector* examples to illustrate several concepts. In all cases, we assume there is already a *Selector* instantiated with a *HtmlResponse* object like this:

```
sel = Selector(html_response)
```

1. Select all `<h1>` elements from an HTML response body, returning a list of *Selector* objects (ie. a *SelectorList* object):

```
sel.xpath("//h1")
```

2. Extract the text of all `<h1>` elements from an HTML response body, returning a list of unicode strings:

```
sel.xpath("//h1").extract()           # this includes the h1 tag
sel.xpath("//h1/text()").extract()    # this excludes the h1 tag
```

3. Iterate over all `<p>` tags and print their class attribute:

```
for node in sel.xpath("//p"):
    print node.xpath("@class").extract()
```

## Selector examples on XML response

Here's a couple of examples to illustrate several concepts. In both cases we assume there is already a *Selector* instantiated with an *XmlResponse* object like this:

```
sel = Selector(xml_response)
```

1. Select all `<product>` elements from an XML response body, returning a list of *Selector* objects (ie. a *SelectorList* object):

```
sel.xpath("//product")
```

2. Extract all prices from a [Google Base XML feed](#) which requires registering a namespace:

```
sel.register_namespace("g", "http://base.google.com/ns/1.0")
sel.xpath("//g:price").extract()
```

## Removing namespaces

When dealing with scraping projects, it is often quite convenient to get rid of namespaces altogether and just work with element names, to write more simple/convenient XPaths. You can use the `Selector.remove_namespaces()` method for that.

Let's show an example that illustrates this with GitHub blog atom feed.

First, we open the shell with the url we want to scrape:

```
$ scrapy shell https://github.com/blog.atom
```

Once in the shell we can try selecting all `<link>` objects and see that it doesn't work (because the Atom XML namespace is obfuscating those nodes):

```
>>> response.xpath("//link")
[]
```

But once we call the `Selector.remove_namespaces()` method, all nodes can be accessed directly by their names:

```
>>> response.selector.remove_namespaces()
>>> response.xpath("//link")
[<Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom">,
<Selector xpath='//link' data=u'<link xmlns="http://www.w3.org/2005/Atom">,
...]
```

If you wonder why the namespace removal procedure isn't always called by default instead of having to call it manually, this is because of two reasons, which, in order of relevance, are:

1. Removing namespaces requires to iterate and modify all nodes in the document, which is a reasonably expensive operation to perform for all documents crawled by Scrapy
2. There could be some cases where using namespaces is actually required, in case some element names clash between namespaces. These cases are very rare though.

## 3.4 Items

The main goal in scraping is to extract structured data from unstructured sources, typically, web pages. Scrapy spiders can return the extracted data as Python dicts. While convenient and familiar, Python dicts lack structure: it is easy to make a typo in a field name or return inconsistent data, especially in a larger project with many spiders.

To define common output data format Scrapy provides the `Item` class. `Item` objects are simple containers used to collect the scraped data. They provide a [dictionary-like](#) API with a convenient syntax for declaring their available fields.

Various Scrapy components use extra information provided by Items: exporters look at declared fields to figure out columns to export, serialization can be customized using Item fields metadata, `trackref` tracks Item instances to help find memory leaks (see [Debugging memory leaks with trackref](#)), etc.

### 3.4.1 Declaring Items

Items are declared using a simple class definition syntax and `Field` objects. Here is an example:

```
import scrapy

class Product(scrapy.Item):
    name = scrapy.Field()
    price = scrapy.Field()
    stock = scrapy.Field()
    last_updated = scrapy.Field(serializer=str)
```

**Note:** Those familiar with [Django](#) will notice that Scrapy Items are declared similar to [Django Models](#), except that Scrapy Items are much simpler as there is no concept of different field types.

### 3.4.2 Item Fields

*Field* objects are used to specify metadata for each field. For example, the serializer function for the `last_updated` field illustrated in the example above.

You can specify any kind of metadata for each field. There is no restriction on the values accepted by *Field* objects. For this same reason, there is no reference list of all available metadata keys. Each key defined in *Field* objects could be used by a different component, and only those components know about it. You can also define and use any other *Field* key in your project too, for your own needs. The main goal of *Field* objects is to provide a way to define all field metadata in one place. Typically, those components whose behaviour depends on each field use certain field keys to configure that behaviour. You must refer to their documentation to see which metadata keys are used by each component.

It's important to note that the *Field* objects used to declare the item do not stay assigned as class attributes. Instead, they can be accessed through the `Item.fields` attribute.

### 3.4.3 Working with Items

Here are some examples of common tasks performed with items, using the `Product` item *declared above*. You will notice the API is very similar to the `dict` API.

#### Creating items

```
>>> product = Product(name='Desktop PC', price=1000)
>>> print product
Product(name='Desktop PC', price=1000)
```

#### Getting field values

```
>>> product['name']
Desktop PC
>>> product.get('name')
Desktop PC

>>> product['price']
1000

>>> product['last_updated']
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
KeyError: 'last_updated'

>>> product.get('last_updated', 'not set')
not set

>>> product['lala'] # getting unknown field
Traceback (most recent call last):
...
KeyError: 'lala'

>>> product.get('lala', 'unknown field')
'unknown field'

>>> 'name' in product # is name field populated?
True

>>> 'last_updated' in product # is last_updated populated?
False

>>> 'last_updated' in product.fields # is last_updated a declared field?
True

>>> 'lala' in product.fields # is lala a declared field?
False
```

## Setting field values

```
>>> product['last_updated'] = 'today'
>>> product['last_updated']
today

>>> product['lala'] = 'test' # setting unknown field
Traceback (most recent call last):
...
KeyError: 'Product does not support field: lala'
```

## Accessing all populated values

To access all populated values, just use the typical `dict` API:

```
>>> product.keys()
['price', 'name']

>>> product.items()
[('price', 1000), ('name', 'Desktop PC')]
```

## Other common tasks

Copying items:

```
>>> product2 = Product(product)
>>> print product2
Product(name='Desktop PC', price=1000)

>>> product3 = product2.copy()
>>> print product3
Product(name='Desktop PC', price=1000)
```

Creating dicts from items:

```
>>> dict(product) # create a dict from all populated values
{'price': 1000, 'name': 'Desktop PC'}
```

Creating items from dicts:

```
>>> Product({'name': 'Laptop PC', 'price': 1500})
Product(price=1500, name='Laptop PC')

>>> Product({'name': 'Laptop PC', 'lala': 1500}) # warning: unknown field in dict
Traceback (most recent call last):
...
KeyError: 'Product does not support field: lala'
```

### 3.4.4 Extending Items

You can extend Items (to add more fields or to change some metadata for some fields) by declaring a subclass of your original Item.

For example:

```
class DiscountedProduct(Product):
    discount_percent = scrapy.Field(serializer=str)
    discount_expiration_date = scrapy.Field()
```

You can also extend field metadata by using the previous field metadata and appending more values, or changing existing values, like this:

```
class SpecificProduct(Product):
    name = scrapy.Field(Product.fields['name'], serializer=my_serializer)
```

That adds (or replaces) the `serializer` metadata key for the `name` field, keeping all the previously existing metadata values.

### 3.4.5 Item objects

```
class scrapy.item.Item([arg])
```

Return a new Item optionally initialized from the given argument.

Items replicate the standard `dict` API, including its constructor. The only additional attribute provided by Items is:

#### **fields**

A dictionary containing *all declared fields* for this Item, not only those populated. The keys are the field names and the values are the *Field* objects used in the *Item declaration*.

### 3.4.6 Field objects

**class** scrapy.item.Field([arg])

The *Field* class is just an alias to the built-in `dict` class and doesn't provide any extra functionality or attributes. In other words, *Field* objects are plain-old Python dicts. A separate class is used to support the *item declaration syntax* based on class attributes.

## 3.5 Item Loaders

Item Loaders provide a convenient mechanism for populating scraped *Items*. Even though Items can be populated using their own dictionary-like API, Item Loaders provide a much more convenient API for populating them from a scraping process, by automating some common tasks like parsing the raw extracted data before assigning it.

In other words, *Items* provide the *container* of scraped data, while Item Loaders provide the mechanism for *populating* that container.

Item Loaders are designed to provide a flexible, efficient and easy mechanism for extending and overriding different field parsing rules, either by spider, or by source format (HTML, XML, etc) without becoming a nightmare to maintain.

### 3.5.1 Using Item Loaders to populate items

To use an Item Loader, you must first instantiate it. You can either instantiate it with a dict-like object (e.g. Item or dict) or without one, in which case an Item is automatically instantiated in the Item Loader constructor using the Item class specified in the `ItemLoader.default_item_class` attribute.

Then, you start collecting values into the Item Loader, typically using *Selectors*. You can add more than one value to the same item field; the Item Loader will know how to “join” those values later using a proper processing function.

Here is a typical Item Loader usage in a *Spider*, using the *Product item* declared in the *Items chapter*:

```
from scrapy.loader import ItemLoader
from myproject.items import Product

def parse(self, response):
    l = ItemLoader(item=Product(), response=response)
    l.add_xpath('name', '//div[@class="product_name"]')
    l.add_xpath('name', '//div[@class="product_title"]')
    l.add_xpath('price', '//p[@id="price"]')
    l.add_css('stock', 'p#stock')
    l.add_value('last_updated', 'today') # you can also use literal values
    return l.load_item()
```

By quickly looking at that code, we can see the `name` field is being extracted from two different XPath locations in the page:

1. `//div[@class="product_name"]`
2. `//div[@class="product_title"]`

In other words, data is being collected by extracting it from two XPath locations, using the `add_xpath()` method. This is the data that will be assigned to the `name` field later.

Afterwards, similar calls are used for `price` and `stock` fields (the latter using a CSS selector with the `add_css()` method), and finally the `last_update` field is populated directly with a literal value (`today`) using a different method: `add_value()`.

Finally, when all data is collected, the `ItemLoader.load_item()` method is called which actually returns the item populated with the data previously extracted and collected with the `add_xpath()`, `add_css()`, and `add_value()` calls.

### 3.5.2 Input and Output processors

An Item Loader contains one input processor and one output processor for each (item) field. The input processor processes the extracted data as soon as it's received (through the `add_xpath()`, `add_css()` or `add_value()` methods) and the result of the input processor is collected and kept inside the ItemLoader. After collecting all data, the `ItemLoader.load_item()` method is called to populate and get the populated `Item` object. That's when the output processor is called with the data previously collected (and processed using the input processor). The result of the output processor is the final value that gets assigned to the item.

Let's see an example to illustrate how the input and output processors are called for a particular field (the same applies for any other field):

```
l = ItemLoader(Product(), some_selector)
l.add_xpath('name', xpath1) # (1)
l.add_xpath('name', xpath2) # (2)
l.add_css('name', css) # (3)
l.add_value('name', 'test') # (4)
return l.load_item() # (5)
```

So what happens is:

1. Data from `xpath1` is extracted, and passed through the *input processor* of the `name` field. The result of the input processor is collected and kept in the Item Loader (but not yet assigned to the item).
2. Data from `xpath2` is extracted, and passed through the same *input processor* used in (1). The result of the input processor is appended to the data collected in (1) (if any).
3. This case is similar to the previous ones, except that the data is extracted from the `css` CSS selector, and passed through the same *input processor* used in (1) and (2). The result of the input processor is appended to the data collected in (1) and (2) (if any).
4. This case is also similar to the previous ones, except that the value to be collected is assigned directly, instead of being extracted from a XPath expression or a CSS selector. However, the value is still passed through the input processors. In this case, since the value is not iterable it is converted to an iterable of a single element before passing it to the input processor, because input processor always receive iterables.
5. The data collected in steps (1), (2), (3) and (4) is passed through the *output processor* of the `name` field. The result of the output processor is the value assigned to the `name` field in the item.

It's worth noticing that processors are just callable objects, which are called with the data to be parsed, and return a parsed value. So you can use any function as input or output processor. The only requirement is that they must accept one (and only one) positional argument, which will be an iterator.

---

**Note:** Both input and output processors must receive an iterator as their first argument. The output of those functions can be anything. The result of input processors will be appended to an internal list (in the Loader) containing the collected values (for that field). The result of the output processors is the value that will be finally assigned to the item.

---

If you want to use a plain function as a processor, make sure it receives `self` as the first argument:

```
def lowercase_processor(self, values):
    for v in values:
        yield v.lower()
```

(continues on next page)

(continued from previous page)

```
class MyItemLoader(ItemLoader):
    name_in = lowercase_processor
```

This is because whenever a function is assigned as a class variable, it becomes a method and would be passed the instance as the the first argument when being called. See [this answer on stackoverflow](#) for more details.

The other thing you need to keep in mind is that the values returned by input processors are collected internally (in lists) and then passed to output processors to populate the fields.

Last, but not least, Scrapy comes with some *commonly used processors* built-in for convenience.

### 3.5.3 Declaring Item Loaders

Item Loaders are declared like Items, by using a class definition syntax. Here is an example:

```
from scrapy.loader import ItemLoader
from scrapy.loader.processors import TakeFirst, MapCompose, Join

class ProductLoader(ItemLoader):

    default_output_processor = TakeFirst()

    name_in = MapCompose(unicode.title)
    name_out = Join()

    price_in = MapCompose(unicode.strip)

    # ...
```

As you can see, input processors are declared using the `_in` suffix while output processors are declared using the `_out` suffix. And you can also declare a default input/output processors using the `ItemLoader.default_input_processor` and `ItemLoader.default_output_processor` attributes.

### 3.5.4 Declaring Input and Output Processors

As seen in the previous section, input and output processors can be declared in the Item Loader definition, and it's very common to declare input processors this way. However, there is one more place where you can specify the input and output processors to use: in the *Item Field* metadata. Here is an example:

```
import scrapy
from scrapy.loader.processors import Join, MapCompose, TakeFirst
from w3lib.html import remove_tags

def filter_price(value):
    if value.isdigit():
        return value

class Product(scrapy.Item):
    name = scrapy.Field(
        input_processor=MapCompose(remove_tags),
        output_processor=Join(),
    )
    price = scrapy.Field(
```

(continues on next page)



(continued from previous page)

```

    input_processor=MapCompose(remove_tags, filter_price),
    output_processor=TakeFirst(),
)

```

```

>>> from scrapy.loader import ItemLoader
>>> il = ItemLoader(item=Product())
>>> il.add_value('name', [u'Welcome to my', u'<strong>website</strong>'])
>>> il.add_value('price', [u'€', u'<span>1000</span>'])
>>> il.load_item()
{'name': u'Welcome to my website', 'price': u'1000'}

```

The precedence order, for both input and output processors, is as follows:

1. Item Loader field-specific attributes: `field_in` and `field_out` (most precedence)
2. Field metadata (input\_processor and output\_processor key)
3. Item Loader defaults: `ItemLoader.default_input_processor()` and `ItemLoader.default_output_processor()` (least precedence)

See also: *Reusing and extending Item Loaders*.

### 3.5.5 Item Loader Context

The Item Loader Context is a dict of arbitrary key/values which is shared among all input and output processors in the Item Loader. It can be passed when declaring, instantiating or using Item Loader. They are used to modify the behaviour of the input/output processors.

For example, suppose you have a function `parse_length` which receives a text value and extracts a length from it:

```

def parse_length(text, loader_context):
    unit = loader_context.get('unit', 'm')
    # ... length parsing code goes here ...
    return parsed_length

```

By accepting a `loader_context` argument the function is explicitly telling the Item Loader that it's able to receive an Item Loader context, so the Item Loader passes the currently active context when calling it, and the processor function (`parse_length` in this case) can thus use them.

There are several ways to modify Item Loader context values:

1. By modifying the currently active Item Loader context (*context* attribute):

```

loader = ItemLoader(product)
loader.context['unit'] = 'cm'

```

2. On Item Loader instantiation (the keyword arguments of Item Loader constructor are stored in the Item Loader context):

```

loader = ItemLoader(product, unit='cm')

```

3. On Item Loader declaration, for those input/output processors that support instantiating them with an Item Loader context. `MapCompose` is one of them:

```

class ProductLoader(ItemLoader):
    length_out = MapCompose(parse_length, unit='cm')

```

### 3.5.6 ItemLoader objects

**class** scrapy.loader.ItemLoader ([*item*, *selector*, *response*], **\*\*kwargs**)

Return a new Item Loader for populating the given Item. If no item is given, one is instantiated automatically using the class in `default_item_class`.

When instantiated with a *selector* or a *response* parameters the *ItemLoader* class provides convenient mechanisms for extracting data from web pages using *selectors*.

#### Parameters

- **item** (*Item* object) – The item instance to populate using subsequent calls to `add_xpath()`, `add_css()`, or `add_value()`.
- **selector** (*Selector* object) – The selector to extract data from, when using the `add_xpath()` (resp. `add_css()`) or `replace_xpath()` (resp. `replace_css()`) method.
- **response** (*Response* object) – The response used to construct the selector using the `default_selector_class`, unless the selector argument is given, in which case this argument is ignored.

The item, selector, response and the remaining keyword arguments are assigned to the Loader context (accessible through the `context` attribute).

*ItemLoader* instances have the following methods:

**get\_value** (*value*, *\*processors*, **\*\*kwargs**)

Process the given value by the given processors and keyword arguments.

Available keyword arguments:

**Parameters** **re** (*str* or *compiled regex*) – a regular expression to use for extracting data from the given value using `extract_regex()` method, applied before processors

Examples:

```
>>> from scrapy.loader.processors import TakeFirst
>>> loader.get_value(u'name: foo', TakeFirst(), unicode.upper, re='name: (.+)')
↪ 'FOO'
```

**add\_value** (*field\_name*, *value*, *\*processors*, **\*\*kwargs**)

Process and then add the given value for the given field.

The value is first passed through `get_value()` by giving the processors and kwargs, and then passed through the *field input processor* and its result appended to the data collected for that field. If the field already contains collected data, the new data is added.

The given *field\_name* can be `None`, in which case values for multiple fields may be added. And the processed value should be a dict with *field\_name* mapped to values.

Examples:

```
loader.add_value('name', u'Color TV')
loader.add_value('colours', [u'white', u'blue'])
loader.add_value('length', u'100')
loader.add_value('name', u'name: foo', TakeFirst(), re='name: (.+)')
loader.add_value(None, {'name': u'foo', 'sex': u'male'})
```

**replace\_value** (*field\_name*, *value*, *\*processors*, **\*\*kwargs**)

Similar to `add_value()` but replaces the collected data with the new value instead of adding it.

**get\_xpath** (*xpath*, *\*processors*, *\*\*kwargs*)

Similar to *ItemLoader.get\_value()* but receives an XPath instead of a value, which is used to extract a list of unicode strings from the selector associated with this *ItemLoader*.

#### Parameters

- **xpath** (*str*) – the XPath to extract data from
- **re** (*str or compiled regex*) – a regular expression to use for extracting data from the selected XPath region

Examples:

```
# HTML snippet: <p class="product-name">Color TV</p>
loader.get_xpath('//p[@class="product-name"]')
# HTML snippet: <p id="price">the price is $1200</p>
loader.get_xpath('//p[@id="price"]', TakeFirst(), re='the price is (.*)')
```

**add\_xpath** (*field\_name*, *xpath*, *\*processors*, *\*\*kwargs*)

Similar to *ItemLoader.add\_value()* but receives an XPath instead of a value, which is used to extract a list of unicode strings from the selector associated with this *ItemLoader*.

See *get\_xpath()* for kwargs.

**Parameters** **xpath** (*str*) – the XPath to extract data from

Examples:

```
# HTML snippet: <p class="product-name">Color TV</p>
loader.add_xpath('name', '//p[@class="product-name"]')
# HTML snippet: <p id="price">the price is $1200</p>
loader.add_xpath('price', '//p[@id="price"]', re='the price is (.*)')
```

**replace\_xpath** (*field\_name*, *xpath*, *\*processors*, *\*\*kwargs*)

Similar to *add\_xpath()* but replaces collected data instead of adding it.

**get\_css** (*css*, *\*processors*, *\*\*kwargs*)

Similar to *ItemLoader.get\_value()* but receives a CSS selector instead of a value, which is used to extract a list of unicode strings from the selector associated with this *ItemLoader*.

#### Parameters

- **css** (*str*) – the CSS selector to extract data from
- **re** (*str or compiled regex*) – a regular expression to use for extracting data from the selected CSS region

Examples:

```
# HTML snippet: <p class="product-name">Color TV</p>
loader.get_css('p.product-name')
# HTML snippet: <p id="price">the price is $1200</p>
loader.get_css('p#price', TakeFirst(), re='the price is (.*)')
```

**add\_css** (*field\_name*, *css*, *\*processors*, *\*\*kwargs*)

Similar to *ItemLoader.add\_value()* but receives a CSS selector instead of a value, which is used to extract a list of unicode strings from the selector associated with this *ItemLoader*.

See *get\_css()* for kwargs.

**Parameters** **css** (*str*) – the CSS selector to extract data from

Examples:

```
# HTML snippet: <p class="product-name">Color TV</p>
loader.add_css('name', 'p.product-name')
# HTML snippet: <p id="price">the price is $1200</p>
loader.add_css('price', 'p#price', re='the price is (.*)')
```

**replace\_css** (*field\_name*, *css*, *\*processors*, *\*\*kwargs*)

Similar to `add_css()` but replaces collected data instead of adding it.

**load\_item** ()

Populate the item with the data collected so far, and return it. The data collected is first passed through the *output processors* to get the final value to assign to each item field.

**nested\_xpath** (*xpath*)

Create a nested loader with an xpath selector. The supplied selector is applied relative to selector associated with this *ItemLoader*. The nested loader shares the *Item* with the parent *ItemLoader* so calls to `add_xpath()`, `add_value()`, `replace_value()`, etc. will behave as expected.

**nested\_css** (*css*)

Create a nested loader with a css selector. The supplied selector is applied relative to selector associated with this *ItemLoader*. The nested loader shares the *Item* with the parent *ItemLoader* so calls to `add_xpath()`, `add_value()`, `replace_value()`, etc. will behave as expected.

**get\_collected\_values** (*field\_name*)

Return the collected values for the given field.

**get\_output\_value** (*field\_name*)

Return the collected values parsed using the output processor, for the given field. This method doesn't populate or modify the item at all.

**get\_input\_processor** (*field\_name*)

Return the input processor for the given field.

**get\_output\_processor** (*field\_name*)

Return the output processor for the given field.

*ItemLoader* instances have the following attributes:

**item**

The *Item* object being parsed by this Item Loader.

**context**

The currently active *Context* of this Item Loader.

**default\_item\_class**

An Item class (or factory), used to instantiate items when not given in the constructor.

**default\_input\_processor**

The default input processor to use for those fields which don't specify one.

**default\_output\_processor**

The default output processor to use for those fields which don't specify one.

**default\_selector\_class**

The class used to construct the *selector* of this *ItemLoader*, if only a response is given in the constructor. If a selector is given in the constructor this attribute is ignored. This attribute is sometimes overridden in subclasses.

**selector**

The *Selector* object to extract data from. It's either the selector given in the constructor or one created from the response given in the constructor using the `default_selector_class`. This attribute is meant to be read-only.

### 3.5.7 Nested Loaders

When parsing related values from a subsection of a document, it can be useful to create nested loaders. Imagine you're extracting details from a footer of a page that looks something like:

Example:

```
<footer>
  <a class="social" href="https://facebook.com/whatever">Like Us</a>
  <a class="social" href="https://twitter.com/whatever">Follow Us</a>
  <a class="email" href="mailto:whatever@example.com">Email Us</a>
</footer>
```

Without nested loaders, you need to specify the full xpath (or css) for each value that you wish to extract.

Example:

```
loader = ItemLoader(item=Item())
# load stuff not in the footer
loader.add_xpath('social', '//footer/a[@class = "social"]/@href')
loader.add_xpath('email', '//footer/a[@class = "email"]/@href')
loader.load_item()
```

Instead, you can create a nested loader with the footer selector and add values relative to the footer. The functionality is the same but you avoid repeating the footer selector.

Example:

```
loader = ItemLoader(item=Item())
# load stuff not in the footer
footer_loader = loader.nested_xpath('//footer')
footer_loader.add_xpath('social', 'a[@class = "social"]/@href')
footer_loader.add_xpath('email', 'a[@class = "email"]/@href')
# no need to call footer_loader.load_item()
loader.load_item()
```

You can nest loaders arbitrarily and they work with either xpath or css selectors. As a general guideline, use nested loaders when they make your code simpler but do not go overboard with nesting or your parser can become difficult to read.

### 3.5.8 Reusing and extending Item Loaders

As your project grows bigger and acquires more and more spiders, maintenance becomes a fundamental problem, especially when you have to deal with many different parsing rules for each spider, having a lot of exceptions, but also wanting to reuse the common processors.

Item Loaders are designed to ease the maintenance burden of parsing rules, without losing flexibility and, at the same time, providing a convenient mechanism for extending and overriding them. For this reason Item Loaders support traditional Python class inheritance for dealing with differences of specific spiders (or groups of spiders).

Suppose, for example, that some particular site encloses their product names in three dashes (e.g. ---Plasma TV---) and you don't want to end up scraping those dashes in the final product names.

Here's how you can remove those dashes by reusing and extending the default Product Item Loader (ProductLoader):

```
from scrapy.loader.processors import MapCompose
from myproject.ItemLoaders import ProductLoader

def strip_dashes(x):
    return x.strip('-')

class SiteSpecificLoader(ProductLoader):
    name_in = MapCompose(strip_dashes, ProductLoader.name_in)
```

Another case where extending Item Loaders can be very helpful is when you have multiple source formats, for example XML and HTML. In the XML version you may want to remove CDATA occurrences. Here's an example of how to do it:

```
from scrapy.loader.processors import MapCompose
from myproject.ItemLoaders import ProductLoader
from myproject.utils.xml import remove_cdata

class XmlProductLoader(ProductLoader):
    name_in = MapCompose(remove_cdata, ProductLoader.name_in)
```

And that's how you typically extend input processors.

As for output processors, it is more common to declare them in the field metadata, as they usually depend only on the field and not on each specific site parsing rule (as input processors do). See also: *Declaring Input and Output Processors*.

There are many other possible ways to extend, inherit and override your Item Loaders, and different Item Loaders hierarchies may fit better for different projects. Scrapy only provides the mechanism; it doesn't impose any specific organization of your Loaders collection - that's up to you and your project's needs.

### 3.5.9 Available built-in processors

Even though you can use any callable function as input and output processors, Scrapy provides some commonly used processors, which are described below. Some of them, like the *MapCompose* (which is typically used as input processor) compose the output of several functions executed in order, to produce the final parsed value.

Here is a list of all built-in processors:

#### **class** scrapy.loader.processors.Identity

The simplest processor, which doesn't do anything. It returns the original values unchanged. It doesn't receive any constructor arguments, nor does it accept Loader contexts.

Example:

```
>>> from scrapy.loader.processors import Identity
>>> proc = Identity()
>>> proc(['one', 'two', 'three'])
['one', 'two', 'three']
```

#### **class** scrapy.loader.processors.TakeFirst

Returns the first non-null/non-empty value from the values received, so it's typically used as an output processor to single-valued fields. It doesn't receive any constructor arguments, nor does it accept Loader contexts.

Example:

```
>>> from scrapy.loader.processors import TakeFirst
>>> proc = TakeFirst()
```

(continues on next page)

(continued from previous page)

```
>>> proc(['', 'one', 'two', 'three'])
'one'
```

**class** scrapy.loader.processors.**Join**(separator=u' ')

Returns the values joined with the separator given in the constructor, which defaults to u' '. It doesn't accept Loader contexts.

When using the default separator, this processor is equivalent to the function: u' '.join

Examples:

```
>>> from scrapy.loader.processors import Join
>>> proc = Join()
>>> proc(['one', 'two', 'three'])
u'one two three'
>>> proc = Join('<br>')
>>> proc(['one', 'two', 'three'])
u'one<br>two<br>three'
```

**class** scrapy.loader.processors.**Compose**(\*functions, \*\*default\_loader\_context)

A processor which is constructed from the composition of the given functions. This means that each input value of this processor is passed to the first function, and the result of that function is passed to the second function, and so on, until the last function returns the output value of this processor.

By default, stop process on None value. This behaviour can be changed by passing keyword argument stop\_on\_none=False.

Example:

```
>>> from scrapy.loader.processors import Compose
>>> proc = Compose(lambda v: v[0], str.upper)
>>> proc(['hello', 'world'])
'HELLO'
```

Each function can optionally receive a loader\_context parameter. For those which do, this processor will pass the currently active *Loader context* through that parameter.

The keyword arguments passed in the constructor are used as the default Loader context values passed to each function call. However, the final Loader context values passed to functions are overridden with the currently active Loader context accessible through the ItemLoader.context() attribute.

**class** scrapy.loader.processors.**MapCompose**(\*functions, \*\*default\_loader\_context)

A processor which is constructed from the composition of the given functions, similar to the *Compose* processor. The difference with this processor is the way internal results are passed among functions, which is as follows:

The input value of this processor is *iterated* and the first function is applied to each element. The results of these function calls (one for each element) are concatenated to construct a new iterable, which is then used to apply the second function, and so on, until the last function is applied to each value of the list of values collected so far. The output values of the last function are concatenated together to produce the output of this processor.

Each particular function can return a value or a list of values, which is flattened with the list of values returned by the same function applied to the other input values. The functions can also return None in which case the output of that function is ignored for further processing over the chain.

This processor provides a convenient way to compose functions that only work with single values (instead of iterables). For this reason the *MapCompose* processor is typically used as input processor, since data is often extracted using the *extract()* method of *selectors*, which returns a list of unicode strings.

The example below should clarify how it works:

```
>>> def filter_world(x):
...     return None if x == 'world' else x
...
>>> from scrapy.loader.processors import MapCompose
>>> proc = MapCompose(filter_world, unicode.upper)
>>> proc([u'hello', u'world', u'this', u'is', u'scrappy'])
[u'HELLO', u'THIS', u'IS', u'SCRAPY']
```

As with the Compose processor, functions can receive Loader contexts, and constructor keyword arguments are used as default context values. See *Compose* processor for more info.

**class** scrapy.loader.processors.**SelectJmes** (*json\_path*)

Queries the value using the json path provided to the constructor and returns the output. Requires *jmespath* (<https://github.com/jmespath/jmespath.py>) to run. This processor takes only one input at a time.

Example:

```
>>> from scrapy.loader.processors import SelectJmes, Compose, MapCompose
>>> proc = SelectJmes("foo") #for direct use on lists and dictionaries
>>> proc({'foo': 'bar'})
'bar'
>>> proc({'foo': {'bar': 'baz'}})
{'bar': 'baz'}
```

Working with Json:

```
>>> import json
>>> proc_single_json_str = Compose(json.loads, SelectJmes("foo"))
>>> proc_single_json_str('{"foo": "bar"}')
u'bar'
>>> proc_json_list = Compose(json.loads, MapCompose(SelectJmes('foo')))
>>> proc_json_list(' [{"foo": "bar"}, {"baz": "tar"} ]')
[u'bar']
```

## 3.6 Scrapy shell

The Scrapy shell is an interactive shell where you can try and debug your scraping code very quickly, without having to run the spider. It's meant to be used for testing data extraction code, but you can actually use it for testing any kind of code as it is also a regular Python shell.

The shell is used for testing XPath or CSS expressions and see how they work and what data they extract from the web pages you're trying to scrape. It allows you to interactively test your expressions while you're writing your spider, without having to run the spider to test every change.

Once you get familiarized with the Scrapy shell, you'll see that it's an invaluable tool for developing and debugging your spiders.

### 3.6.1 Configuring the shell

If you have *IPython* installed, the Scrapy shell will use it (instead of the standard Python console). The *IPython* console is much more powerful and provides smart auto-completion and colorized output, among other things.

We highly recommend you install *IPython*, specially if you're working on Unix systems (where *IPython* excels). See the *IPython installation guide* for more info.



Scrapy also has support for `bpython`, and will try to use it where `IPython` is unavailable.

Through scrapy's settings you can configure it to use any one of `ipython`, `bpython` or the standard `python` shell, regardless of which are installed. This is done by setting the `SCRAPY_PYTHON_SHELL` environment variable; or by defining it in your `scrapy.cfg`:

```
[settings]
shell = bpython
```

### 3.6.2 Launch the shell

To launch the Scrapy shell you can use the **shell** command like this:

```
scrapy shell <url>
```

Where the `<url>` is the URL you want to scrape.

**shell** also works for local files. This can be handy if you want to play around with a local copy of a web page. **shell** understands the following syntaxes for local files:

```
# UNIX-style
scrapy shell ./path/to/file.html
scrapy shell ../other/path/to/file.html
scrapy shell /absolute/path/to/file.html

# File URI
scrapy shell file:///absolute/path/to/file.html
```

**Note:** When using relative file paths, be explicit and prepend them with `./` (or `../` when relevant). `scrapy shell index.html` will not work as one might expect (and this is by design, not a bug).

Because **shell** favors HTTP URLs over File URIs, and `index.html` being syntactically similar to `example.com`, **shell** will treat `index.html` as a domain name and trigger a DNS lookup error:

```
$ scrapy shell index.html
[ ... scrapy shell starts ... ]
[ ... traceback ... ]
twisted.internet.error.DNSLookupError: DNS lookup failed:
address 'index.html' not found: [Errno -5] No address associated with hostname.
```

**shell** will not test beforehand if a file called `index.html` exists in the current directory. Again, be explicit.

### 3.6.3 Using the shell

The Scrapy shell is just a regular Python console (or `IPython` console if you have it available) which provides some additional shortcut functions for convenience.

#### Available Shortcuts

- `shelp()` - print a help with the list of available objects and shortcuts

- `fetch(url[, redirect=True])` - fetch a new response from the given URL and update all related objects accordingly. You can optionally ask for HTTP 3xx redirections to not be followed by passing `redirect=False`
- `fetch(request)` - fetch a new response from the given request and update all related objects accordingly.
- `view(response)` - open the given response in your local web browser, for inspection. This will add a `<base>` tag to the response body in order for external links (such as images and style sheets) to display properly. Note, however, that this will create a temporary file in your computer, which won't be removed automatically.

## Available Scrapy objects

The Scrapy shell automatically creates some convenient objects from the downloaded page, like the `Response` object and the `Selector` objects (for both HTML and XML content).

Those objects are:

- `crawler` - the current `Crawler` object.
- `spider` - the Spider which is known to handle the URL, or a `Spider` object if there is no spider found for the current URL
- `request` - a `Request` object of the last fetched page. You can modify this request using `replace()` or fetch a new request (without leaving the shell) using the `fetch` shortcut.
- `response` - a `Response` object containing the last fetched page
- `settings` - the current `Scrapy settings`

## 3.6.4 Example of shell session

Here's an example of a typical shell session where we start by scraping the <https://scrapy.org> page, and then proceed to scrape the <https://reddit.com> page. Finally, we modify the (Reddit) request method to POST and re-fetch it getting an error. We end the session by typing Ctrl-D (in Unix systems) or Ctrl-Z in Windows.

Keep in mind that the data extracted here may not be the same when you try it, as those pages are not static and could have changed by the time you test this. The only purpose of this example is to get you familiarized with how the Scrapy shell works.

First, we launch the shell:

```
scrapy shell 'https://scrapy.org' --nolog
```

Then, the shell fetches the URL (using the Scrapy downloader) and prints the list of available objects and useful shortcuts (you'll notice that these lines all start with the `[s]` prefix):

```
[s] Available Scrapy objects:
[s] scrapy      scrapy module (contains scrapy.Request, scrapy.Selector, etc)
[s] crawler    <scrapy.crawler.Crawler object at 0x7f07395dd690>
[s] item       {}
[s] request    <GET https://scrapy.org>
[s] response   <200 https://scrapy.org/>
[s] settings   <scrapy.settings.Settings object at 0x7f07395dd710>
[s] spider     <DefaultSpider 'default' at 0x7f0735891690>
[s] Useful shortcuts:
[s] fetch(url[, redirect=True]) Fetch URL and update local objects (by default, ↵
↵ redirects are followed)
[s] fetch(req)                  Fetch a scrapy.Request and update local objects
```

(continues on next page)

(continued from previous page)

```
[s]  shelp()          Shell help (print this help)
[s]  view(response)   View response in a browser

>>>
```

After that, we can start playing with the objects:

```
>>> response.xpath('//title/text()').extract_first()
'Scrappy | A Fast and Powerful Scraping and Web Crawling Framework'

>>> fetch("https://reddit.com")

>>> response.xpath('//title/text()').extract()
['reddit: the front page of the internet']

>>> request = request.replace(method="POST")

>>> fetch(request)

>>> response.status
404

>>> from pprint import pprint

>>> pprint(response.headers)
{'Accept-Ranges': ['bytes'],
 'Cache-Control': ['max-age=0, must-revalidate'],
 'Content-Type': ['text/html; charset=UTF-8'],
 'Date': ['Thu, 08 Dec 2016 16:21:19 GMT'],
 'Server': ['snooserv'],
 'Set-Cookie': ['loid=KgNLou0V9SKMX4qb4n; Domain=reddit.com; Max-Age=63071999; Path=/;
↳ expires=Sat, 08-Dec-2018 16:21:19 GMT; secure',
               'loidcreated=2016-12-08T16%3A21%3A19.445Z; Domain=reddit.com; Max-
↳ Age=63071999; Path=/; expires=Sat, 08-Dec-2018 16:21:19 GMT; secure',
               'loid=vi0ZVe4NkxNWdlH7r7; Domain=reddit.com; Max-Age=63071999; Path=/;
↳ expires=Sat, 08-Dec-2018 16:21:19 GMT; secure',
               'loidcreated=2016-12-08T16%3A21%3A19.459Z; Domain=reddit.com; Max-
↳ Age=63071999; Path=/; expires=Sat, 08-Dec-2018 16:21:19 GMT; secure'],
 'Vary': ['accept-encoding'],
 'Via': ['1.1 varnish'],
 'X-Cache': ['MISS'],
 'X-Cache-Hits': ['0'],
 'X-Content-Type-Options': ['nosniff'],
 'X-Frame-Options': ['SAMEORIGIN'],
 'X-Moose': ['majestic'],
 'X-Served-By': ['cache-cdg8730-CDG'],
 'X-Timer': ['S1481214079.394283,VS0,VE159'],
 'X-Ua-Compatible': ['IE=edge'],
 'X-Xss-Protection': ['1; mode=block']}
>>>
```

### 3.6.5 Invoking the shell from spiders to inspect responses

Sometimes you want to inspect the responses that are being processed in a certain point of your spider, if only to check that response you expect is getting there.

This can be achieved by using the `scrapy.shell.inspect_response` function.

Here's an example of how you would call it from your spider:

```
import scrapy

class MySpider(scrapy.Spider):
    name = "myspider"
    start_urls = [
        "http://example.com",
        "http://example.org",
        "http://example.net",
    ]

    def parse(self, response):
        # We want to inspect one specific response.
        if ".org" in response.url:
            from scrapy.shell import inspect_response
            inspect_response(response, self)

        # Rest of parsing code.
```

When you run the spider, you will get something similar to this:

```
2014-01-23 17:48:31-0400 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://
↪example.com> (referer: None)
2014-01-23 17:48:31-0400 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://
↪example.org> (referer: None)
[s] Available Scrapy objects:
[s]   crawler      <scrapy.crawler.Crawler object at 0x1e16b50>
...

>>> response.url
'http://example.org'
```

Then, you can check if the extraction code is working:

```
>>> response.xpath('//h1[@class="fn"]')
[]
```

Nope, it doesn't. So you can open the response in your web browser and see if it's the response you were expecting:

```
>>> view(response)
True
```

Finally you hit Ctrl-D (or Ctrl-Z in Windows) to exit the shell and resume the crawling:

```
>>> ^D
2014-01-23 17:50:03-0400 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://
↪example.net> (referer: None)
...
```

Note that you can't use the `fetch` shortcut here since the Scrapy engine is blocked by the shell. However, after you leave the shell, the spider will continue crawling where it stopped, as shown above.

## 3.7 Item Pipeline

After an item has been scraped by a spider, it is sent to the Item Pipeline which processes it through several components that are executed sequentially.

Each item pipeline component (sometimes referred as just “Item Pipeline”) is a Python class that implements a simple method. They receive an item and perform an action over it, also deciding if the item should continue through the pipeline or be dropped and no longer processed.

Typical uses of item pipelines are:

- cleansing HTML data
- validating scraped data (checking that the items contain certain fields)
- checking for duplicates (and dropping them)
- storing the scraped item in a database

### 3.7.1 Writing your own item pipeline

Each item pipeline component is a Python class that must implement the following method:

**process\_item** (*self*, *item*, *spider*)

This method is called for every item pipeline component. `process_item()` must either: return a dict with data, return an *Item* (or any descendant class) object, return a *Twisted Deferred* or raise *DropItem* exception. Dropped items are no longer processed by further pipeline components.

**Parameters**

- **item** (*Item* object or a dict) – the item scraped
- **spider** (*Spider* object) – the spider which scraped the item

Additionally, they may also implement the following methods:

**open\_spider** (*self*, *spider*)

This method is called when the spider is opened.

**Parameters** **spider** (*Spider* object) – the spider which was opened

**close\_spider** (*self*, *spider*)

This method is called when the spider is closed.

**Parameters** **spider** (*Spider* object) – the spider which was closed

**from\_crawler** (*cls*, *crawler*)

If present, this classmethod is called to create a pipeline instance from a *Crawler*. It must return a new instance of the pipeline. Crawler object provides access to all Scrapy core components like settings and signals; it is a way for pipeline to access them and hook its functionality into Scrapy.

**Parameters** **crawler** (*Crawler* object) – crawler that uses this pipeline

### 3.7.2 Item pipeline example

#### Price validation and dropping items with no prices

Let’s take a look at the following hypothetical pipeline that adjusts the `price` attribute for those items that do not include VAT (`price_excludes_vat` attribute), and drops those items which don’t contain a price:

```
from scrapy.exceptions import DropItem

class PricePipeline(object):

    vat_factor = 1.15

    def process_item(self, item, spider):
        if item['price']:
            if item['price_excludes_vat']:
                item['price'] = item['price'] * self.vat_factor
            return item
        else:
            raise DropItem("Missing price in %s" % item)
```

### Write items to a JSON file

The following pipeline stores all scraped items (from all spiders) into a single `items.jsonl` file, containing one item per line serialized in JSON format:

```
import json

class JsonWriterPipeline(object):

    def open_spider(self, spider):
        self.file = open('items.jsonl', 'w')

    def close_spider(self, spider):
        self.file.close()

    def process_item(self, item, spider):
        line = json.dumps(dict(item)) + "\n"
        self.file.write(line)
        return item
```

---

**Note:** The purpose of `JsonWriterPipeline` is just to introduce how to write item pipelines. If you really want to store all scraped items into a JSON file you should use the *Feed exports*.

---

### Write items to MongoDB

In this example we'll write items to [MongoDB](#) using [pymongo](#). MongoDB address and database name are specified in Scrapy settings; MongoDB collection is named after item class.

The main point of this example is to show how to use `from_crawler()` method and how to clean up the resources properly.:

```
import pymongo

class MongoPipeline(object):

    collection_name = 'scrapy_items'

    def __init__(self, mongo_uri, mongo_db):
```

(continues on next page)

(continued from previous page)

```

self.mongo_uri = mongo_uri
self.mongo_db = mongo_db

@classmethod
def from_crawler(cls, crawler):
    return cls(
        mongo_uri=crawler.settings.get('MONGO_URI'),
        mongo_db=crawler.settings.get('MONGO_DATABASE', 'items')
    )

def open_spider(self, spider):
    self.client = pymongo.MongoClient(self.mongo_uri)
    self.db = self.client[self.mongo_db]

def close_spider(self, spider):
    self.client.close()

def process_item(self, item, spider):
    self.db[self.collection_name].insert_one(dict(item))
    return item

```

### Take screenshot of item

This example demonstrates how to return `Deferred` from `process_item()` method. It uses `Splash` to render screenshot of item url. Pipeline makes request to locally running instance of `Splash`. After request is downloaded and `Deferred` callback fires, it saves item to a file and adds filename to an item.

```

import scrapy
import hashlib
from urllib.parse import quote

class ScreenshotPipeline(object):
    """Pipeline that uses Splash to render screenshot of
    every Scrapy item."""

    SPLASH_URL = "http://localhost:8050/render.png?url={}"

    def process_item(self, item, spider):
        encoded_item_url = quote(item["url"])
        screenshot_url = self.SPLASH_URL.format(encoded_item_url)
        request = scrapy.Request(screenshot_url)
        dfd = spider.crawler.engine.download(request, spider)
        dfd.addBoth(self.return_item, item)
        return dfd

    def return_item(self, response, item):
        if response.status != 200:
            # Error happened, return item.
            return item

        # Save screenshot to file, filename will be hash of url.
        url = item["url"]
        url_hash = hashlib.md5(url.encode("utf8")).hexdigest()
        filename = "{}.png".format(url_hash)

```

(continues on next page)

(continued from previous page)

```
with open(filename, "wb") as f:
    f.write(response.body)

# Store filename in item.
item["screenshot_filename"] = filename
return item
```

## Duplicates filter

A filter that looks for duplicate items, and drops those items that were already processed. Let's say that our items have a unique id, but our spider returns multiples items with the same id:

```
from scrapy.exceptions import DropItem

class DuplicatesPipeline(object):

    def __init__(self):
        self.ids_seen = set()

    def process_item(self, item, spider):
        if item['id'] in self.ids_seen:
            raise DropItem("Duplicate item found: %s" % item)
        else:
            self.ids_seen.add(item['id'])
            return item
```

### 3.7.3 Activating an Item Pipeline component

To activate an Item Pipeline component you must add its class to the **:setting:'ITEM\_PIPELINES'** setting, like in the following example:

```
ITEM_PIPELINES = {
    'myproject.pipelines.PricePipeline': 300,
    'myproject.pipelines.JsonWriterPipeline': 800,
}
```

The integer values you assign to classes in this setting determine the order in which they run: items go through from lower valued to higher valued classes. It's customary to define these numbers in the 0-1000 range.

## 3.8 Feed exports

New in version 0.10.

One of the most frequently required features when implementing scrapers is being able to store the scraped data properly and, quite often, that means generating an “export file” with the scraped data (commonly called “export feed”) to be consumed by other systems.

Scrapy provides this functionality out of the box with the Feed Exports, which allows you to generate a feed with the scraped items, using multiple serialization formats and storage backends.



### 3.8.1 Serialization formats

For serializing the scraped data, the feed exports use the *Item exporters*. These formats are supported out of the box:

- *JSON*
- *JSON lines*
- *CSV*
- *XML*

But you can also extend the supported format through the **:setting:'FEED\_EXPORTERS'** setting.

#### JSON

- **:setting:'FEED\_FORMAT':** json
- Exporter used: *JsonItemExporter*
- See *this warning* if you're using JSON with large feeds.

#### JSON lines

- **:setting:'FEED\_FORMAT':** jsonlines
- Exporter used: *JsonLinesItemExporter*

#### CSV

- **:setting:'FEED\_FORMAT':** csv
- Exporter used: *CsvItemExporter*
- To specify columns to export and their order use **:setting:'FEED\_EXPORT\_FIELDS'**. Other feed exporters can also use this option, but it is important for CSV because unlike many other export formats CSV uses a fixed header.

#### XML

- **:setting:'FEED\_FORMAT':** xml
- Exporter used: *XmlItemExporter*

#### Pickle

- **:setting:'FEED\_FORMAT':** pickle
- Exporter used: *PickleItemExporter*

#### Marshal

- **:setting:'FEED\_FORMAT':** marshal
- Exporter used: *MarshalItemExporter*

### 3.8.2 Storages

When using the feed exports you define where to store the feed using a [URI](#) (through the **:setting:‘FEED\_URI‘** setting). The feed exports supports multiple storage backend types which are defined by the URI scheme.

The storages backends supported out of the box are:

- *Local filesystem*
- *FTP*
- *S3* (requires [botocore](#) or [boto](#))
- *Standard output*

Some storage backends may be unavailable if the required external libraries are not available. For example, the S3 backend is only available if the [botocore](#) or [boto](#) library is installed (Scrapy supports [boto](#) only on Python 2).

### 3.8.3 Storage URI parameters

The storage URI can also contain parameters that get replaced when the feed is being created. These parameters are:

- `%(time)s` - gets replaced by a timestamp when the feed is being created
- `%(name)s` - gets replaced by the spider name

Any other named parameter gets replaced by the spider attribute of the same name. For example, `%(site_id)s` would get replaced by the `spider.site_id` attribute the moment the feed is being created.

Here are some examples to illustrate:

- Store in FTP using one directory per spider:
  - `ftp://user:password@ftp.example.com/scraping/feeds/%(name)s/%(time)s.json`
- Store in S3 using one directory per spider:
  - `s3://mybucket/scraping/feeds/%(name)s/%(time)s.json`

### 3.8.4 Storage backends

#### Local filesystem

The feeds are stored in the local filesystem.

- URI scheme: `file`
- Example URI: `file:///tmp/export.csv`
- Required external libraries: `none`

Note that for the local filesystem storage (only) you can omit the scheme if you specify an absolute path like `/tmp/export.csv`. This only works on Unix systems though.

#### FTP

The feeds are stored in a FTP server.

- URI scheme: `ftp`

- Example URI: `ftp://user:pass@ftp.example.com/path/to/export.csv`
- Required external libraries: none

## S3

The feeds are stored on [Amazon S3](#).

- URI scheme: `s3`
- Example URIs:
  - `s3://mybucket/path/to/export.csv`
  - `s3://aws_key:aws_secret@mybucket/path/to/export.csv`
- Required external libraries: [botocore](#) (Python 2 and Python 3) or [boto](#) (Python 2 only)

The AWS credentials can be passed as user/password in the URI, or they can be passed through the following settings:

- `:setting:'AWS_ACCESS_KEY_ID'`
- `:setting:'AWS_SECRET_ACCESS_KEY'`

## Standard output

The feeds are written to the standard output of the Scrapy process.

- URI scheme: `stdout`
- Example URI: `stdout:`
- Required external libraries: none

## 3.8.5 Settings

These are the settings used for configuring the feed exports:

- `:setting:'FEED_URI'` (mandatory)
- `:setting:'FEED_FORMAT'`
- `:setting:'FEED_STORAGES'`
- `:setting:'FEED_EXPORTERS'`
- `:setting:'FEED_STORE_EMPTY'`
- `:setting:'FEED_EXPORT_ENCODING'`
- `:setting:'FEED_EXPORT_FIELDS'`
- `:setting:'FEED_EXPORT_INDENT'`

### FEED\_URI

Default: None

The URI of the export feed. See [Storage backends](#) for supported URI schemes.

This setting is required for enabling the feed exports.

## FEED\_FORMAT

The serialization format to be used for the feed. See *Serialization formats* for possible values.

## FEED\_EXPORT\_ENCODING

Default: `None`

The encoding to be used for the feed.

If unset or set to `None` (default) it uses UTF-8 for everything except JSON output, which uses safe numeric encoding (`\uXXXX` sequences) for historic reasons.

Use `utf-8` if you want UTF-8 for JSON too.

## FEED\_EXPORT\_FIELDS

Default: `None`

A list of fields to export, optional. Example: `FEED_EXPORT_FIELDS = ["foo", "bar", "baz"]`.

Use `FEED_EXPORT_FIELDS` option to define fields to export and their order.

When `FEED_EXPORT_FIELDS` is empty or `None` (default), Scrapy uses fields defined in dicts or *Item* subclasses a spider is yielding.

If an exporter requires a fixed set of fields (this is the case for *CSV* export format) and `FEED_EXPORT_FIELDS` is empty or `None`, then Scrapy tries to infer field names from the exported data - currently it uses field names from the first item.

## FEED\_EXPORT\_INDENT

Default: `0`

Amount of spaces used to indent the output on each level. If `FEED_EXPORT_INDENT` is a non-negative integer, then array elements and object members will be pretty-printed with that indent level. An indent level of `0` (the default), or negative, will put each item on a new line. `None` selects the most compact representation.

Currently implemented only by *JsonItemExporter* and *XmlItemExporter*, i.e. when you are exporting to `.json` or `.xml`.

## FEED\_STORE\_EMPTY

Default: `False`

Whether to export empty feeds (ie. feeds with no items).

## FEED\_STORAGES

Default: `{}`

A dict containing additional feed storage backends supported by your project. The keys are URI schemes and the values are paths to storage classes.

## FEED\_STORAGES\_BASE

Default:

```
{
    '': 'scrapy.extensions.feedexport.FileFeedStorage',
    'file': 'scrapy.extensions.feedexport.FileFeedStorage',
    'stdout': 'scrapy.extensions.feedexport.StdoutFeedStorage',
    's3': 'scrapy.extensions.feedexport.S3FeedStorage',
    'ftp': 'scrapy.extensions.feedexport.FTPFeedStorage',
}
```

A dict containing the built-in feed storage backends supported by Scrapy. You can disable any of these backends by assigning `None` to their URI scheme in **:setting:'FEED\_STORAGES'**. E.g., to disable the built-in FTP storage backend (without replacement), place this in your `settings.py`:

```
FEED_STORAGES = {
    'ftp': None,
}
```

## FEED\_EXPORTERS

Default: {}

A dict containing additional exporters supported by your project. The keys are serialization formats and the values are paths to *Item exporter* classes.

## FEED\_EXPORTERS\_BASE

Default:

```
{
    'json': 'scrapy.exporters.JsonItemExporter',
    'jsonlines': 'scrapy.exporters.JsonLinesItemExporter',
    'jl': 'scrapy.exporters.JsonLinesItemExporter',
    'csv': 'scrapy.exporters.CsvItemExporter',
    'xml': 'scrapy.exporters.XmlItemExporter',
    'marshal': 'scrapy.exporters.MarshalItemExporter',
    'pickle': 'scrapy.exporters.PickleItemExporter',
}
```

A dict containing the built-in feed exporters supported by Scrapy. You can disable any of these exporters by assigning `None` to their serialization format in **:setting:'FEED\_EXPORTERS'**. E.g., to disable the built-in CSV exporter (without replacement), place this in your `settings.py`:

```
FEED_EXPORTERS = {
    'csv': None,
}
```

## 3.9 Requests and Responses

Scrapy uses *Request* and *Response* objects for crawling web sites.

Typically, *Request* objects are generated in the spiders and pass across the system until they reach the Downloader, which executes the request and returns a *Response* object which travels back to the spider that issued the request.

Both *Request* and *Response* classes have subclasses which add functionality not required in the base classes. These are described below in *Request subclasses* and *Response subclasses*.

### 3.9.1 Request objects

```
class scrapy.http.Request(url[, callback, method='GET', headers, body, cookies, meta,
                           encoding='utf-8', priority=0, dont_filter=False, errback, flags])
```

A *Request* object represents an HTTP request, which is usually generated in the Spider and executed by the Downloader, and thus generating a *Response*.

#### Parameters

- **url** (*string*) – the URL of this request
- **callback** (*callable*) – the function that will be called with the response of this request (once its downloaded) as its first parameter. For more information see *Passing additional data to callback functions* below. If a Request doesn't specify a callback, the spider's *parse()* method will be used. Note that if exceptions are raised during processing, *errback* is called instead.
- **method** (*string*) – the HTTP method of this request. Defaults to 'GET'.
- **meta** (*dict*) – the initial values for the *Request.meta* attribute. If given, the dict passed in this parameter will be shallow copied.
- **body** (*str or unicode*) – the request body. If a unicode is passed, then it's encoded to *str* using the *encoding* passed (which defaults to *utf-8*). If *body* is not given, an empty string is stored. Regardless of the type of this argument, the final value stored will be a *str* (never unicode or *None*).
- **headers** (*dict*) – the headers of this request. The dict values can be strings (for single valued headers) or lists (for multi-valued headers). If *None* is passed as value, the HTTP header will not be sent at all.
- **cookies** (*dict or list*) – the request cookies. These can be sent in two forms.

1. Using a dict:

```
request_with_cookies = Request(url="http://www.example.com",
                               cookies={'currency': 'USD',
                                         ↳ 'country': 'UY'})
```

2. Using a list of dicts:

```
request_with_cookies = Request(url="http://www.example.com",
                               cookies=[{'name': 'currency',
                                         'value': 'USD',
                                         'domain': 'example.com',
                                         'path': '/currency'}])
```

The latter form allows for customizing the *domain* and *path* attributes of the cookie. This is only useful if the cookies are saved for later requests.

When some site returns cookies (in a response) those are stored in the cookies for that domain and will be sent again in future requests. That's the typical behaviour of any regular web browser. However, if, for some reason, you want to avoid merging with existing cookies

you can instruct Scrapy to do so by setting the `dont_merge_cookies` key to `True` in the `Request.meta`.

Example of request without merging cookies:

```
request_with_cookies = Request(url="http://www.example.com",
                               cookies={'currency': 'USD', 'country': 'UY'},
                               meta={'dont_merge_cookies': True})
```

For more info see [CookiesMiddleware](#).

- **encoding** (*string*) – the encoding of this request (defaults to `'utf-8'`). This encoding will be used to percent-encode the URL and to convert the body to `str` (if given as unicode).
- **priority** (*int*) – the priority of this request (defaults to 0). The priority is used by the scheduler to define the order used to process requests. Requests with a higher priority value will execute earlier. Negative values are allowed in order to indicate relatively low-priority.
- **dont\_filter** (*boolean*) – indicates that this request should not be filtered by the scheduler. This is used when you want to perform an identical request multiple times, to ignore the duplicates filter. Use it with care, or you will get into crawling loops. Default to `False`.
- **errback** (*callable*) – a function that will be called if any exception was raised while processing the request. This includes pages that failed with 404 HTTP errors and such. It receives a `Twisted Failure` instance as first parameter. For more information, see [Using errbacks to catch exceptions in request processing](#) below.
- **flags** (*list*) – Flags sent to the request, can be used for logging or similar purposes.

#### url

A string containing the URL of this request. Keep in mind that this attribute contains the escaped URL, so it can differ from the URL passed in the constructor.

This attribute is read-only. To change the URL of a Request use [replace\(\)](#).

#### method

A string representing the HTTP method in the request. This is guaranteed to be uppercase. Example: `"GET"`, `"POST"`, `"PUT"`, etc

#### headers

A dictionary-like object which contains the request headers.

#### body

A str that contains the request body.

This attribute is read-only. To change the body of a Request use [replace\(\)](#).

#### meta

A dict that contains arbitrary metadata for this request. This dict is empty for new Requests, and is usually populated by different Scrapy components (extensions, middlewares, etc). So the data contained in this dict depends on the extensions you have enabled.

See [Request.meta special keys](#) for a list of special meta keys recognized by Scrapy.

This dict is [shallow copied](#) when the request is cloned using the `copy()` or `replace()` methods, and can also be accessed, in your spider, from the `response.meta` attribute.

#### copy()

Return a new Request which is a copy of this Request. See also: [Passing additional data to callback functions](#).

**replace** ([url, method, headers, body, cookies, meta, encoding, dont\_filter, callback, errback])

Return a Request object with the same members, except for those members given new values by whichever keyword arguments are specified. The attribute `Request.meta` is copied by default (unless a new value is given in the `meta` argument). See also *Passing additional data to callback functions*.

### Passing additional data to callback functions

The callback of a request is a function that will be called when the response of that request is downloaded. The callback function will be called with the downloaded `Response` object as its first argument.

Example:

```
def parse_page1(self, response):
    return scrapy.Request("http://www.example.com/some_page.html",
                           callback=self.parse_page2)

def parse_page2(self, response):
    # this would log http://www.example.com/some_page.html
    self.logger.info("Visited %s", response.url)
```

In some cases you may be interested in passing arguments to those callback functions so you can receive the arguments later, in the second callback. You can use the `Request.meta` attribute for that.

Here's an example of how to pass an item using this mechanism, to populate different fields from different pages:

```
def parse_page1(self, response):
    item = MyItem()
    item['main_url'] = response.url
    request = scrapy.Request("http://www.example.com/some_page.html",
                              callback=self.parse_page2)
    request.meta['item'] = item
    yield request

def parse_page2(self, response):
    item = response.meta['item']
    item['other_url'] = response.url
    yield item
```

### Using errbacks to catch exceptions in request processing

The errback of a request is a function that will be called when an exception is raised while processing it.

It receives a `Twisted Failure` instance as first parameter and can be used to track connection establishment timeouts, DNS errors etc.

Here's an example spider logging all errors and catching some specific errors if needed:

```
import scrapy

from scrapy.spidermiddlewares.httperror import HttpError
from twisted.internet.error import DNSLookupError
from twisted.internet.error import TimeoutError, TCTimedOutError

class ErrbackSpider(scrapy.Spider):
    name = "errback_example"
    start_urls = [
```

(continues on next page)



(continued from previous page)

```

        "http://www.httpbin.org/",          # HTTP 200 expected
        "http://www.httpbin.org/status/404", # Not found error
        "http://www.httpbin.org/status/500", # server issue
        "http://www.httpbin.org:12345/",    # non-responding host, timeout_
→expected
        "http://www.httphttpbinbin.org/",   # DNS error expected
    ]

    def start_requests(self):
        for u in self.start_urls:
            yield scrapy.Request(u, callback=self.parse_httpbin,
                                errback=self.errback_httpbin,
                                dont_filter=True)

    def parse_httpbin(self, response):
        self.logger.info('Got successful response from {}'.format(response.url))
        # do something useful here...

    def errback_httpbin(self, failure):
        # log all failures
        self.logger.error(repr(failure))

        # in case you want to do something special for some errors,
        # you may need the failure's type:

        if failure.check(HttpError):
            # these exceptions come from HttpError spider middleware
            # you can get the non-200 response
            response = failure.value.response
            self.logger.error('HttpError on %s', response.url)

        elif failure.check(DNSLookupError):
            # this is the original request
            request = failure.request
            self.logger.error('DNSLookupError on %s', request.url)

        elif failure.check(TimeoutError, TCPTimedOutError):
            request = failure.request
            self.logger.error('TimeoutError on %s', request.url)

```

### 3.9.2 Request.meta special keys

The `Request.meta` attribute can contain any arbitrary data, but there are some special keys recognized by Scrapy and its built-in extensions.

Those are:

- `:reqmeta:'dont_redirect'`
- `:reqmeta:'dont_retry'`
- `:reqmeta:'handle_httpstatus_list'`
- `:reqmeta:'handle_httpstatus_all'`
- `:reqmeta:'dont_merge_cookies'`
- `:reqmeta:'cookiejar'`

- `:reqmeta:'dont_cache'`
- `:reqmeta:'redirect_urls'`
- `:reqmeta:'bindaddress'`
- `:reqmeta:'dont_obey_robotstxt'`
- `:reqmeta:'download_timeout'`
- `:reqmeta:'download_maxsize'`
- `:reqmeta:'download_latency'`
- `:reqmeta:'download_fail_on_datatoss'`
- `:reqmeta:'proxy'`
- `ftp_user` (See `:setting:'FTP_USER'` for more info)
- `ftp_password` (See `:setting:'FTP_PASSWORD'` for more info)
- `:reqmeta:'referrer_policy'`
- `:reqmeta:'max_retry_times'`

#### `bindaddress`

The IP of the outgoing IP address to use for the performing the request.

#### `download_timeout`

The amount of time (in secs) that the downloader will wait before timing out. See also: `:setting:'DOWNLOAD_TIMEOUT'`.

#### `download_latency`

The amount of time spent to fetch the response, since the request has been started, i.e. HTTP message sent over the network. This meta key only becomes available when the response has been downloaded. While most other meta keys are used to control Scrapy behavior, this one is supposed to be read-only.

#### `download_fail_on_datatoss`

Whether or not to fail on broken responses. See: `:setting:'DOWNLOAD_FAIL_ON_DATALOSS'`.

#### `max_retry_times`

The meta key is used set retry times per request. When initialized, the `:reqmeta:'max_retry_times'` meta key takes higher precedence over the `:setting:'RETRY_TIMES'` setting.

### 3.9.3 Request subclasses

Here is the list of built-in *Request* subclasses. You can also subclass it to implement your own custom functionality.

## FormRequest objects

The `FormRequest` class extends the base `Request` with functionality for dealing with HTML forms. It uses `lxml.html` forms to pre-populate form fields with form data from `Response` objects.

```
class scrapy.http.FormRequest(url[, formdata, ...])
```

The `FormRequest` class adds a new argument to the constructor. The remaining arguments are the same as for the `Request` class and are not documented here.

**Parameters** `formdata` (*dict or iterable of tuples*) – is a dictionary (or iterable of (key, value) tuples) containing HTML Form data which will be url-encoded and assigned to the body of the request.

The `FormRequest` objects support the following class method in addition to the standard `Request` methods:

```
classmethod from_response(response[, formname=None, formid=None, formnumber=0, formdata=None, formxpath=None, formcss=None, clickdata=None, dont_click=False, ...])
```

Returns a new `FormRequest` object with its form field values pre-populated with those found in the HTML `<form>` element contained in the given response. For an example see [Using FormRequest.from\\_response\(\) to simulate a user login](#).

The policy is to automatically simulate a click, by default, on any form control that looks clickable, like a `<input type="submit">`. Even though this is quite convenient, and often the desired behaviour, sometimes it can cause problems which could be hard to debug. For example, when working with forms that are filled and/or submitted using javascript, the default `from_response()` behaviour may not be the most appropriate. To disable this behaviour you can set the `dont_click` argument to `True`. Also, if you want to change the control clicked (instead of disabling it) you can also use the `clickdata` argument.

**Caution:** Using this method with select elements which have leading or trailing whitespace in the option values will not work due to a [bug in lxml](#), which should be fixed in `lxml 3.8` and above.

### Parameters

- **response** (`Response` object) – the response containing a HTML form which will be used to pre-populate the form fields
- **formname** (`string`) – if given, the form with name attribute set to this value will be used.
- **formid** (`string`) – if given, the form with id attribute set to this value will be used.
- **formxpath** (`string`) – if given, the first form that matches the xpath will be used.
- **formcss** (`string`) – if given, the first form that matches the css selector will be used.
- **formnumber** (`integer`) – the number of form to use, when the response contains multiple forms. The first one (and also the default) is 0.
- **formdata** (`dict`) – fields to override in the form data. If a field was already present in the response `<form>` element, its value is overridden by the one passed in this parameter. If a value passed in this parameter is `None`, the field will not be included in the request, even if it was present in the response `<form>` element.
- **clickdata** (`dict`) – attributes to lookup the control clicked. If it's not given, the form data will be submitted simulating a click on the first clickable element. In addition to html attributes, the control can be identified by its zero-based index relative to other submittable inputs inside the form, via the `nr` attribute.

- **dont\_click** (*boolean*) – If True, the form data will be submitted without clicking in any element.

The other parameters of this class method are passed directly to the *FormRequest* constructor.

New in version 0.10.3: The *formname* parameter.

New in version 0.17: The *formxpath* parameter.

New in version 1.1.0: The *formcss* parameter.

New in version 1.1.0: The *formid* parameter.

## Request usage examples

### Using FormRequest to send data via HTTP POST

If you want to simulate a HTML Form POST in your spider and send a couple of key-value fields, you can return a *FormRequest* object (from your spider) like this:

```
return [FormRequest(url="http://www.example.com/post/action",
                    formdata={'name': 'John Doe', 'age': '27'},
                    callback=self.after_post)]
```

### Using FormRequest.from\_response() to simulate a user login

It is usual for web sites to provide pre-populated form fields through `<input type="hidden">` elements, such as session related data or authentication tokens (for login pages). When scraping, you'll want these fields to be automatically pre-populated and only override a couple of them, such as the user name and password. You can use the *FormRequest.from\_response()* method for this job. Here's an example spider which uses it:

```
import scrapy

class LoginSpider(scrapy.Spider):
    name = 'example.com'
    start_urls = ['http://www.example.com/users/login.php']

    def parse(self, response):
        return scrapy.FormRequest.from_response(
            response,
            formdata={'username': 'john', 'password': 'secret'},
            callback=self.after_login
        )

    def after_login(self, response):
        # check login succeed before going on
        if "authentication failed" in response.body:
            self.logger.error("Login failed")
            return

        # continue scraping with authenticated session...
```

### 3.9.4 Response objects

**class** scrapy.http.**Response** (*url*[, *status*=200, *headers*=None, *body*=b'', *flags*=None, *request*=None])

A *Response* object represents an HTTP response, which is usually downloaded (by the Downloader) and fed to the Spiders for processing.

#### Parameters

- **url** (*string*) – the URL of this response
- **status** (*integer*) – the HTTP status of the response. Defaults to 200.
- **headers** (*dict*) – the headers of this response. The dict values can be strings (for single valued headers) or lists (for multi-valued headers).
- **body** (*bytes*) – the response body. To access the decoded text as str (unicode in Python 2) you can use `response.text` from an encoding-aware *Response subclass*, such as *TextResponse*.
- **flags** (*list*) – is a list containing the initial values for the *Response.flags* attribute. If given, the list will be shallow copied.
- **request** (*Request* object) – the initial value of the *Response.request* attribute. This represents the *Request* that generated this response.

#### url

A string containing the URL of the response.

This attribute is read-only. To change the URL of a Response use *replace()*.

#### status

An integer representing the HTTP status of the response. Example: 200, 404.

#### headers

A dictionary-like object which contains the response headers. Values can be accessed using *get()* to return the first header value with the specified name or *getlist()* to return all header values with the specified name. For example, this call will give you all cookies in the headers:

```
response.headers.getlist('Set-Cookie')
```

#### body

The body of this Response. Keep in mind that *Response.body* is always a bytes object. If you want the unicode version use *TextResponse.text* (only available in *TextResponse* and subclasses).

This attribute is read-only. To change the body of a Response use *replace()*.

#### request

The *Request* object that generated this response. This attribute is assigned in the Scrapy engine, after the response and the request have passed through all *Downloader Middlewares*. In particular, this means that:

- HTTP redirections will cause the original request (to the URL before redirection) to be assigned to the redirected response (with the final URL after redirection).
- *Response.request.url* doesn't always equal *Response.url*
- This attribute is only available in the spider code, and in the *Spider Middlewares*, but not in Downloader Middlewares (although you have the Request available there by other means) and handlers of the **:signal:'response\_downloaded'** signal.

**meta**

A shortcut to the `Request.meta` attribute of the `Response.request` object (ie. `self.request.meta`).

Unlike the `Response.request` attribute, the `Response.meta` attribute is propagated along redirects and retries, so you will get the original `Request.meta` sent from your spider.

**See also:**

`Request.meta` attribute

**flags**

A list that contains flags for this response. Flags are labels used for tagging Responses. For example: `'cached'`, `'redirected'`, etc. And they're shown on the string representation of the Response (`__str__` method) which is used by the engine for logging.

**copy()**

Returns a new Response which is a copy of this Response.

**replace([url, status, headers, body, request, flags, cls])**

Returns a Response object with the same members, except for those members given new values by whichever keyword arguments are specified. The attribute `Response.meta` is copied by default.

**urljoin(url)**

Constructs an absolute url by combining the Response's `url` with a possible relative url.

This is a wrapper over `urlparse.urljoin`, it's merely an alias for making this call:

```
urlparse.urljoin(response.url, url)
```

### 3.9.5 Response subclasses

Here is the list of available built-in Response subclasses. You can also subclass the Response class to implement your own functionality.

#### TextResponse objects

**class** scrapy.http.**TextResponse**(url[, encoding[, ...]])

`TextResponse` objects adds encoding capabilities to the base `Response` class, which is meant to be used only for binary data, such as images, sounds or any media file.

`TextResponse` objects support a new constructor argument, in addition to the base `Response` objects. The remaining functionality is the same as for the `Response` class and is not documented here.

**Parameters** `encoding` (*string*) – is a string which contains the encoding to use for this response. If you create a `TextResponse` object with a unicode body, it will be encoded using this encoding (remember the body attribute is always a string). If `encoding` is `None` (default value), the encoding will be looked up in the response headers and body instead.

`TextResponse` objects support the following attributes in addition to the standard `Response` ones:

**text**

Response body, as unicode.

The same as `response.body.decode(response.encoding)`, but the result is cached after the first call, so you can access `response.text` multiple times without extra overhead.

---

**Note:** `unicode(response.body)` is not a correct way to convert response body to unicode: you would be using the system default encoding (typically *ascii*) instead of the response encoding.

---

**encoding**

A string with the encoding of this response. The encoding is resolved by trying the following mechanisms, in order:

1. the encoding passed in the constructor *encoding* argument
2. the encoding declared in the Content-Type HTTP header. If this encoding is not valid (ie. unknown), it is ignored and the next resolution mechanism is tried.
3. the encoding declared in the response body. The `TextResponse` class doesn't provide any special functionality for this. However, the `HtmlResponse` and `XmlResponse` classes do.
4. the encoding inferred by looking at the response body. This is the more fragile method but also the last one tried.

**selector**

A `Selector` instance using the response as target. The selector is lazily instantiated on first access.

`TextResponse` objects support the following methods in addition to the standard `Response` ones:

**xpath** (*query*)

A shortcut to `TextResponse.selector.xpath(query)`:

```
response.xpath('//p')
```

**css** (*query*)

A shortcut to `TextResponse.selector.css(query)`:

```
response.css('p')
```

**body\_as\_unicode** ()

The same as `text`, but available as a method. This method is kept for backwards compatibility; please prefer `response.text`.

## HtmlResponse objects

**class** scrapy.http.HtmlResponse (*url*[, ... ])

The `HtmlResponse` class is a subclass of `TextResponse` which adds encoding auto-discovering support by looking into the HTML *meta http-equiv* attribute. See `TextResponse.encoding`.

## XmlResponse objects

**class** scrapy.http.XmlResponse (*url*[, ... ])

The `XmlResponse` class is a subclass of `TextResponse` which adds encoding auto-discovering support by looking into the XML declaration line. See `TextResponse.encoding`.

## 3.10 Link Extractors

Link extractors are objects whose only purpose is to extract links from web pages (`scrapy.http.Response` objects) which will be eventually followed.

There is `scrapy.linkextractors.LinkExtractor` available in Scrapy, but you can create your own custom Link Extractors to suit your needs by implementing a simple interface.

The only public method that every link extractor has is `extract_links`, which receives a *Response* object and returns a list of `scrapy.link.Link` objects. Link extractors are meant to be instantiated once and their `extract_links` method called several times with different responses to extract links to follow.

Link extractors are used in the *CrawlSpider* class (available in Scrapy), through a set of rules, but you can also use it in your spiders, even if you don't subclass from *CrawlSpider*, as its purpose is very simple: to extract links.

### 3.10.1 Built-in link extractors reference

Link extractors classes bundled with Scrapy are provided in the `scrapy.linkextractors` module.

The default link extractor is `LinkExtractor`, which is the same as *LxmlLinkExtractor*:

```
from scrapy.linkextractors import LinkExtractor
```

There used to be other link extractor classes in previous Scrapy versions, but they are deprecated now.

#### LxmlLinkExtractor

```
class scrapy.linkextractors.lxmlhtml.LxmlLinkExtractor (allow=(), deny=(),
                                                         allow_domains=(),
                                                         deny_domains=(),
                                                         deny_extensions=None,
                                                         restrict_xpaths=(), re-
                                                         strict_css=(), tags=('a',
                                                         'area'), attrs=('href',
                                                         ), canonicalize=False,
                                                         unique=True, pro-
                                                         cess_value=None,
                                                         strip=True)
```

*LxmlLinkExtractor* is the recommended link extractor with handy filtering options. It is implemented using *lxml*'s robust *HTMLParser*.

#### Parameters

- **allow** (*a regular expression (or list of)*) – a single regular expression (or list of regular expressions) that the (absolute) urls must match in order to be extracted. If not given (or empty), it will match all links.
- **deny** (*a regular expression (or list of)*) – a single regular expression (or list of regular expressions) that the (absolute) urls must match in order to be excluded (ie. not extracted). It has precedence over the `allow` parameter. If not given (or empty) it won't exclude any links.
- **allow\_domains** (*str or list*) – a single value or a list of string containing domains which will be considered for extracting the links
- **deny\_domains** (*str or list*) – a single value or a list of strings containing domains which won't be considered for extracting the links
- **deny\_extensions** (*list*) – a single value or list of strings containing extensions that should be ignored when extracting links. If not given, it will default to the `IGNORED_EXTENSIONS` list defined in the `scrapy.linkextractors` package.



- **restrict\_xpaths** (*str or list*) – is an XPath (or list of XPath's) which defines regions inside the response where links should be extracted from. If given, only the text selected by those XPath will be scanned for links. See examples below.
- **restrict\_css** (*str or list*) – a CSS selector (or list of selectors) which defines regions inside the response where links should be extracted from. Has the same behaviour as `restrict_xpaths`.
- **tags** (*str or list*) – a tag or a list of tags to consider when extracting links. Defaults to ('a', 'area').
- **attrs** (*list*) – an attribute or list of attributes which should be considered when looking for links to extract (only for those tags specified in the `tags` parameter). Defaults to ('href',)
- **canonicalize** (*boolean*) – canonicalize each extracted url (using `w3lib.url.canonicalize_url`). Defaults to `False`. Note that `canonicalize_url` is meant for duplicate checking; it can change the URL visible at server side, so the response can be different for requests with canonicalized and raw URLs. If you're using `LinkExtractor` to follow links it is more robust to keep the default `canonicalize=False`.
- **unique** (*boolean*) – whether duplicate filtering should be applied to extracted links.
- **process\_value** (*callable*) – a function which receives each value extracted from the tag and attributes scanned and can modify the value and return a new one, or return `None` to ignore the link altogether. If not given, `process_value` defaults to `lambda x: x`.

For example, to extract links from this code:

```
<a href="javascript:goToPage('../other/page.html'); return false">
↪Link text</a>
```

You can use the following function in `process_value`:

```
def process_value(value):
    m = re.search("javascript:goToPage\('(.*?)'", value)
    if m:
        return m.group(1)
```

- **strip** (*boolean*) – whether to strip whitespaces from extracted attributes. According to HTML5 standard, leading and trailing whitespaces must be stripped from `href` attributes of `<a>`, `<area>` and many other elements, `src` attribute of `<img>`, `<iframe>` elements, etc., so `LinkExtractor` strips space chars by default. Set `strip=False` to turn it off (e.g. if you're extracting urls from elements or attributes which allow leading/trailing whitespaces).

## 3.11 Settings

The Scrapy settings allows you to customize the behaviour of all Scrapy components, including the core, extensions, pipelines and spiders themselves.

The infrastructure of the settings provides a global namespace of key-value mappings that the code can use to pull configuration values from. The settings can be populated through different mechanisms, which are described below.

The settings are also the mechanism for selecting the currently active Scrapy project (in case you have many).

For a list of available built-in settings see: [Built-in settings reference](#).

### 3.11.1 Designating the settings

When you use Scrapy, you have to tell it which settings you're using. You can do this by using an environment variable, `SCRAPY_SETTINGS_MODULE`.

The value of `SCRAPY_SETTINGS_MODULE` should be in Python path syntax, e.g. `myproject.settings`. Note that the settings module should be on the Python [import search path](#).

### 3.11.2 Populating the settings

Settings can be populated using different mechanisms, each of which having a different precedence. Here is the list of them in decreasing order of precedence:

1. Command line options (most precedence)
2. Settings per-spider
3. Project settings module
4. Default settings per-command
5. Default global settings (less precedence)

The population of these settings sources is taken care of internally, but a manual handling is possible using API calls. See the [Settings API](#) topic for reference.

These mechanisms are described in more detail below.

#### 1. Command line options

Arguments provided by the command line are the ones that take most precedence, overriding any other options. You can explicitly override one (or more) settings using the `-s` (or `--set`) command line option.

Example:

```
scrapy crawl myspider -s LOG_FILE=scrapy.log
```

#### 2. Settings per-spider

Spiders (See the [Spiders](#) chapter for reference) can define their own settings that will take precedence and override the project ones. They can do so by setting their `custom_settings` attribute:

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

    custom_settings = {
        'SOME_SETTING': 'some value',
    }
```

#### 3. Project settings module

The project settings module is the standard configuration file for your Scrapy project, it's where most of your custom settings will be populated. For a standard Scrapy project, this means you'll be adding or changing the settings in the `settings.py` file created for your project.

## 4. Default settings per-command

Each *Scrapy tool* command can have its own default settings, which override the global default settings. Those custom command settings are specified in the `default_settings` attribute of the command class.

## 5. Default global settings

The global defaults are located in the `scrapy.settings.default_settings` module and documented in the *Built-in settings reference* section.

### 3.11.3 How to access settings

In a spider, the settings are available through `self.settings`:

```
class MySpider(scrapy.Spider):
    name = 'myspider'
    start_urls = ['http://example.com']

    def parse(self, response):
        print("Existing settings: %s" % self.settings.attributes.keys())
```

**Note:** The `settings` attribute is set in the base `Spider` class after the spider is initialized. If you want to use the settings before the initialization (e.g., in your spider's `__init__()` method), you'll need to override the `from_crawler()` method.

Settings can be accessed through the `scrapy.crawler.Crawler.settings` attribute of the `Crawler` that is passed to `from_crawler` method in extensions, middlewares and item pipelines:

```
class MyExtension(object):
    def __init__(self, log_is_enabled=False):
        if log_is_enabled:
            print("log is enabled!")

    @classmethod
    def from_crawler(cls, crawler):
        settings = crawler.settings
        return cls(settings.getbool('LOG_ENABLED'))
```

The settings object can be used like a dict (e.g., `settings['LOG_ENABLED']`), but it's usually preferred to extract the setting in the format you need it to avoid type errors, using one of the methods provided by the `Settings` API.

### 3.11.4 Rationale for setting names

Setting names are usually prefixed with the component that they configure. For example, proper setting names for a fictional `robots.txt` extension would be `ROBOTSTXT_ENABLED`, `ROBOTSTXT_OBEY`, `ROBOTSTXT_CACHEDIR`, etc.

### 3.11.5 Built-in settings reference

Here's a list of all available Scrapy settings, in alphabetical order, along with their default values and the scope where they apply.

The scope, where available, shows where the setting is being used, if it's tied to any particular component. In that case the module of that component will be shown, typically an extension, middleware or pipeline. It also means that the component must be enabled in order for the setting to have any effect.

### AWS\_ACCESS\_KEY\_ID

Default: None

The AWS access key used by code that requires access to [Amazon Web services](#), such as the *S3 feed storage backend*.

### AWS\_SECRET\_ACCESS\_KEY

Default: None

The AWS secret key used by code that requires access to [Amazon Web services](#), such as the *S3 feed storage backend*.

### AWS\_ENDPOINT\_URL

Default: None

Endpoint URL used for S3-like self-hosted storage. Storage like Minio or s3.scality.

### AWS\_USE\_SSL

Default: None

Use this option if you want to disable SSL connection for communication with S3 or S3-like storage. By default SSL will be used.

### AWS\_VERIFY

Default: None

Verify SSL connection between Scrapy and S3 or S3-like storage. By default SSL verification will occur.

### BOT\_NAME

Default: 'scrapybot'

The name of the bot implemented by this Scrapy project (also known as the project name). This will be used to construct the User-Agent by default, and also for logging.

It's automatically populated with your project name when you create your project with the **startproject** command.

### CONCURRENT\_ITEMS

Default: 100

Maximum number of concurrent items (per response) to process in parallel in the Item Processor (also known as the *Item Pipeline*).

## CONCURRENT\_REQUESTS

Default: 16

The maximum number of concurrent (ie. simultaneous) requests that will be performed by the Scrapy downloader.

## CONCURRENT\_REQUESTS\_PER\_DOMAIN

Default: 8

The maximum number of concurrent (ie. simultaneous) requests that will be performed to any single domain.

See also: *AutoThrottle extension* and its **:setting:‘AUTOTHROTTLE\_TARGET\_CONCURRENCY’** option.

## CONCURRENT\_REQUESTS\_PER\_IP

Default: 0

The maximum number of concurrent (ie. simultaneous) requests that will be performed to any single IP. If non-zero, the **:setting:‘CONCURRENT\_REQUESTS\_PER\_DOMAIN’** setting is ignored, and this one is used instead. In other words, concurrency limits will be applied per IP, not per domain.

This setting also affects **:setting:‘DOWNLOAD\_DELAY’** and *AutoThrottle extension*: if **:setting:‘CONCURRENT\_REQUESTS\_PER\_IP’** is non-zero, download delay is enforced per IP, not per domain.

## DEFAULT\_ITEM\_CLASS

Default: 'scrapy.item.Item'

The default class that will be used for instantiating items in the *the Scrapy shell*.

## DEFAULT\_REQUEST\_HEADERS

Default:

```
{
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
    'Accept-Language': 'en',
}
```

The default headers used for Scrapy HTTP Requests. They're populated in the *DefaultHeadersMiddleware*.

## DEPTH\_LIMIT

Default: 0

Scope: scrapy.spidermiddlewares.depth.DepthMiddleware

The maximum depth that will be allowed to crawl for any site. If zero, no limit will be imposed.

## DEPTH\_PRIORITY

Default: 0

Scope: `scrapy.spidermiddlewares.depth.DepthMiddleware`

An integer that is used to adjust the request priority based on its depth:

- if zero (default), no priority adjustment is made from depth
- **a positive value will decrease the priority, i.e. higher depth requests will be processed later** ; this is commonly used when doing breadth-first crawls (BFO)
- a negative value will increase priority, i.e., higher depth requests will be processed sooner (DFO)

See also: *Does Scrapy crawl in breadth-first or depth-first order?* about tuning Scrapy for BFO or DFO.

---

**Note:** This setting adjusts priority **in the opposite way** compared to other priority settings **:setting:'REDIRECT\_PRIORITY\_ADJUST'** and **:setting:'RETRY\_PRIORITY\_ADJUST'**.

---

## DEPTH\_STATS\_VERBOSE

Default: False

Scope: `scrapy.spidermiddlewares.depth.DepthMiddleware`

Whether to collect verbose depth stats. If this is enabled, the number of requests for each depth is collected in the stats.

## DNSCACHE\_ENABLED

Default: True

Whether to enable DNS in-memory cache.

## DNSCACHE\_SIZE

Default: 10000

DNS in-memory cache size.

## DNS\_TIMEOUT

Default: 60

Timeout for processing of DNS queries in seconds. Float is supported.

## DOWNLOADER

Default: `'scrapy.core.downloader.Downloader'`

The downloader to use for crawling.

## DOWNLOADER\_HTTPCLIENTFACTORY

Default: `'scrapy.core.downloader.webclient.ScrapyHTTPClientFactory'`

Defines a Twisted `protocol.ClientFactory` class to use for HTTP/1.0 connections (for `HTTP10DownloadHandler`).

---

**Note:** HTTP/1.0 is rarely used nowadays so you can safely ignore this setting, unless you use Twisted<11.1, or if you really want to use HTTP/1.0 and override **:setting:'DOWNLOAD\_HANDLERS\_BASE'** for `http(s)` scheme accordingly, i.e. to `'scrapy.core.downloader.handlers.http.HTTP10DownloadHandler'`.

---

## DOWNLOADER\_CLIENTCONTEXTFACTORY

Default: `'scrapy.core.downloader.contextfactory.ScrapyClientContextFactory'`

Represents the classpath to the ContextFactory to use.

Here, “ContextFactory” is a Twisted term for SSL/TLS contexts, defining the TLS/SSL protocol version to use, whether to do certificate verification, or even enable client-side authentication (and various other things).

---

**Note:** Scrapy default context factory **does NOT perform remote server certificate verification**. This is usually fine for web scraping.

If you do need remote server certificate verification enabled, Scrapy also has another context factory class that you can set, `'scrapy.core.downloader.contextfactory.BrowserLikeContextFactory'`, which uses the platform’s certificates to validate remote endpoints. **This is only available if you use Twisted>=14.0.**

---

If you do use a custom ContextFactory, make sure it accepts a `method` parameter at init (this is the `OpenSSL.SSL` method mapping **:setting:'DOWNLOADER\_CLIENT\_TLS\_METHOD'**).

## DOWNLOADER\_CLIENT\_TLS\_METHOD

Default: `'TLS'`

Use this setting to customize the TLS/SSL method used by the default HTTP/1.1 downloader.

This setting must be one of these string values:

- `'TLS'`: maps to OpenSSL’s `TLS_method()` (a.k.a `SSLv23_method()`), which allows protocol negotiation, starting from the highest supported by the platform; **default, recommended**
- `'TLSv1.0'`: this value forces HTTPS connections to use TLS version 1.0 ; set this if you want the behavior of Scrapy<1.1
- `'TLSv1.1'`: forces TLS version 1.1
- `'TLSv1.2'`: forces TLS version 1.2
- `'SSLv3'`: forces SSL version 3 (**not recommended**)

---

**Note:** We recommend that you use `PyOpenSSL>=0.13` and `Twisted>=0.13` or above (`Twisted>=14.0` if you can).

---

## DOWNLOADER\_MIDDLEWARES

Default:: {}

A dict containing the downloader middlewares enabled in your project, and their orders. For more info see [Activating a downloader middleware](#).

## DOWNLOADER\_MIDDLEWARES\_BASE

Default:

```
{
    'scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware': 100,
    'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware': 300,
    'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware': 350,
    'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware': 400,
    'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': 500,
    'scrapy.downloadermiddlewares.retry.RetryMiddleware': 550,
    'scrapy.downloadermiddlewares.ajaxcrawl.AjaxCrawlMiddleware': 560,
    'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware': 580,
    'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware': 590,
    'scrapy.downloadermiddlewares.redirect.RedirectMiddleware': 600,
    'scrapy.downloadermiddlewares.cookies.CookiesMiddleware': 700,
    'scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware': 750,
    'scrapy.downloadermiddlewares.stats.DownloaderStats': 850,
    'scrapy.downloadermiddlewares.httpcache.HttpCacheMiddleware': 900,
}
```

A dict containing the downloader middlewares enabled by default in Scrapy. Low orders are closer to the engine, high orders are closer to the downloader. You should never modify this setting in your project, modify **:setting:'DOWNLOADER\_MIDDLEWARES'** instead. For more info see [Activating a downloader middleware](#).

## DOWNLOADER\_STATS

Default: True

Whether to enable downloader stats collection.

## DOWNLOAD\_DELAY

Default: 0

The amount of time (in secs) that the downloader should wait before downloading consecutive pages from the same website. This can be used to throttle the crawling speed to avoid hitting servers too hard. Decimal numbers are supported. Example:

```
DOWNLOAD_DELAY = 0.25      # 250 ms of delay
```

This setting is also affected by the **:setting:'RANDOMIZE\_DOWNLOAD\_DELAY'** setting (which is enabled by default). By default, Scrapy doesn't wait a fixed amount of time between requests, but uses a random interval between  $0.5 * \text{:setting:'DOWNLOAD\_DELAY'}$  and  $1.5 * \text{:setting:'DOWNLOAD\_DELAY'}$ .

When **:setting:'CONCURRENT\_REQUESTS\_PER\_IP'** is non-zero, delays are enforced per ip address instead of per domain.

You can also change this setting per spider by setting `download_delay` spider attribute.



## DOWNLOAD\_HANDLERS

Default: {}

A dict containing the request downloader handlers enabled in your project. See **:setting:‘DOWNLOAD\_HANDLERS\_BASE‘** for example format.

## DOWNLOAD\_HANDLERS\_BASE

Default:

```
{
    'file': 'scrapy.core.downloader.handlers.file.FileDownloadHandler',
    'http': 'scrapy.core.downloader.handlers.http.HTTPDownloadHandler',
    'https': 'scrapy.core.downloader.handlers.http.HTTPDownloadHandler',
    's3': 'scrapy.core.downloader.handlers.s3.S3DownloadHandler',
    'ftp': 'scrapy.core.downloader.handlers.ftp.FTPDownloadHandler',
}
```

A dict containing the request download handlers enabled by default in Scrapy. You should never modify this setting in your project, modify **:setting:‘DOWNLOAD\_HANDLERS‘** instead.

You can disable any of these download handlers by assigning None to their URI scheme in **:setting:‘DOWNLOAD\_HANDLERS‘**. E.g., to disable the built-in FTP handler (without replacement), place this in your settings.py:

```
DOWNLOAD_HANDLERS = {
    'ftp': None,
}
```

## DOWNLOAD\_TIMEOUT

Default: 180

The amount of time (in secs) that the downloader will wait before timing out.

---

**Note:** This timeout can be set per spider using `download_timeout` spider attribute and per-request using **:requestmeta:‘download\_timeout‘** Request.meta key.

---

## DOWNLOAD\_MAXSIZE

Default: *1073741824* (1024MB)

The maximum response size (in bytes) that downloader will download.

If you want to disable it set to 0.

---

**Note:** This size can be set per spider using `download_maxsize` spider attribute and per-request using **:requestmeta:‘download\_maxsize‘** Request.meta key.

---

This feature needs Twisted >= 11.1.

---

## DOWNLOAD\_WARN\_SIZE

Default: 33554432 (32MB)

The response size (in bytes) that downloader will start to warn.

If you want to disable it set to 0.

---

**Note:** This size can be set per spider using `download_warn_size` spider attribute and per-request using `:reqmeta:'download_warn_size'` Request.meta key.

This feature needs Twisted >= 11.1.

---

## DOWNLOAD\_FAIL\_ON\_DATA\_LOSS

Default: True

Whether or not to fail on broken responses, that is, declared Content-Length does not match content sent by the server or chunked response was not properly finish. If True, these responses raise a `ResponseFailed([_DataLoss])` error. If False, these responses are passed through and the flag `data_loss` is added to the response, i.e.: `'data_loss'` in `response.flags` is True.

Optionally, this can be set per-request basis by using the `:reqmeta:'download_fail_on_data_loss'` Request.meta key to False.

---

**Note:** A broken response, or data loss error, may happen under several circumstances, from server misconfiguration to network errors to data corruption. It is up to the user to decide if it makes sense to process broken responses considering they may contain partial or incomplete content. If `:setting:'RETRY_ENABLED'` is True and this setting is set to True, the `ResponseFailed([_DataLoss])` failure will be retried as usual.

---

## DUPEFILTER\_CLASS

Default: `'scrapy.dupefilters.RFPDupeFilter'`

The class used to detect and filter duplicate requests.

The default (`RFPDupeFilter`) filters based on request fingerprint using the `scrapy.utils.request.request_fingerprint` function. In order to change the way duplicates are checked you could subclass `RFPDupeFilter` and override its `request_fingerprint` method. This method should accept scrapy `Request` object and return its fingerprint (a string).

You can disable filtering of duplicate requests by setting `:setting:'DUPEFILTER_CLASS'` to `'scrapy.dupefilters.BaseDupeFilter'`. Be very careful about this however, because you can get into crawling loops. It's usually a better idea to set the `dont_filter` parameter to True on the specific `Request` that should not be filtered.

## DUPEFILTER\_DEBUG

Default: False

By default, `RFPDupeFilter` only logs the first duplicate request. Setting `:setting:'DUPEFILTER_DEBUG'` to True will make it log all duplicate requests.

## EDITOR

Default: `vi` (on Unix systems) or the IDLE editor (on Windows)

The editor to use for editing spiders with the `edit` command. Additionally, if the `EDITOR` environment variable is set, the `edit` command will prefer it over the default setting.

## EXTENSIONS

Default:: `{}`

A dict containing the extensions enabled in your project, and their orders.

## EXTENSIONS\_BASE

Default:

```
{
    'scrapy.extensions.corestats.CoreStats': 0,
    'scrapy.extensions.telnet.TelnetConsole': 0,
    'scrapy.extensions.memusage.MemoryUsage': 0,
    'scrapy.extensions.memdebug.MemoryDebugger': 0,
    'scrapy.extensions.closespider.CloseSpider': 0,
    'scrapy.extensions.feedexport.FeedExporter': 0,
    'scrapy.extensions.logstats.LogStats': 0,
    'scrapy.extensions.spiderstate.SpiderState': 0,
    'scrapy.extensions.throttle.AutoThrottle': 0,
}
```

A dict containing the extensions available by default in Scrapy, and their orders. This setting contains all stable built-in extensions. Keep in mind that some of them need to be enabled through a setting.

For more information See the *extensions user guide* and the *list of available extensions*.

## FEED\_TEMPDIR

The Feed Temp dir allows you to set a custom folder to save crawler temporary files before uploading with *FTP feed storage* and *Amazon S3*.

## FTP\_PASSIVE\_MODE

Default: `True`

Whether or not to use passive mode when initiating FTP transfers.

## FTP\_PASSWORD

Default: `"guest"`

The password to use for FTP connections when there is no `"ftp_password"` in Request meta.

---

**Note:** Paraphrasing [RFC 1635](#), although it is common to use either the password “guest” or one’s e-mail address for anonymous FTP, some FTP servers explicitly ask for the user’s e-mail address and will not allow login with the “guest” password.

---

## FTP\_USER

Default: "anonymous"

The username to use for FTP connections when there is no "ftp\_user" in Request meta.

## ITEM\_PIPELINES

Default: {}

A dict containing the item pipelines to use, and their orders. Order values are arbitrary, but it is customary to define them in the 0-1000 range. Lower orders process before higher orders.

Example:

```
ITEM_PIPELINES = {
    'mybot.pipelines.validate.ValidateMyItem': 300,
    'mybot.pipelines.validate.StoreMyItem': 800,
}
```

## ITEM\_PIPELINES\_BASE

Default: {}

A dict containing the pipelines enabled by default in Scrapy. You should never modify this setting in your project, modify **:setting:‘ITEM\_PIPELINES’** instead.

## LOG\_ENABLED

Default: True

Whether to enable logging.

## LOG\_ENCODING

Default: 'utf-8'

The encoding to use for logging.

## LOG\_FILE

Default: None

File name to use for logging output. If None, standard error will be used.

## LOG\_FORMAT

Default: `'%(asctime)s [%(name)s] %(levelname)s: %(message)s'`

String for formatting log messages. Refer to the [Python logging documentation](#) for the whole list of available placeholders.

## LOG\_DATEFORMAT

Default: `'%Y-%m-%d %H:%M:%S'`

String for formatting date/time, expansion of the `%(asctime)s` placeholder in **:setting:‘LOG\_FORMAT’**. Refer to the [Python datetime documentation](#) for the whole list of available directives.

## LOG\_LEVEL

Default: `'DEBUG'`

Minimum level to log. Available levels are: CRITICAL, ERROR, WARNING, INFO, DEBUG. For more info see [Logging](#).

## LOG\_STDOUT

Default: `False`

If `True`, all standard output (and error) of your process will be redirected to the log. For example if you print `'hello'` it will appear in the Scrapy log.

## LOG\_SHORT\_NAMES

Default: `False`

If `True`, the logs will just contain the root path. If it is set to `False` then it displays the component responsible for the log output

## MEMDEBUG\_ENABLED

Default: `False`

Whether to enable memory debugging.

## MEMDEBUG\_NOTIFY

Default: `[]`

When memory debugging is enabled a memory report will be sent to the specified addresses if this setting is not empty, otherwise the report will be written to the log.

Example:

```
MEMDEBUG_NOTIFY = ['user@example.com']
```

## MEMUSAGE\_ENABLED

Default: True

Scope: scrapy.extensions.memusage

Whether to enable the memory usage extension. This extension keeps track of a peak memory used by the process (it writes it to stats). It can also optionally shutdown the Scrapy process when it exceeds a memory limit (see **:setting:'MEMUSAGE\_LIMIT\_MB'**), and notify by email when that happened (see **:setting:'MEMUSAGE\_NOTIFY\_MAIL'**).

See *Memory usage extension*.

## MEMUSAGE\_LIMIT\_MB

Default: 0

Scope: scrapy.extensions.memusage

The maximum amount of memory to allow (in megabytes) before shutting down Scrapy (if MEMUSAGE\_ENABLED is True). If zero, no check will be performed.

See *Memory usage extension*.

## MEMUSAGE\_CHECK\_INTERVAL\_SECONDS

New in version 1.1.

Default: 60.0

Scope: scrapy.extensions.memusage

The *Memory usage extension* checks the current memory usage, versus the limits set by **:setting:'MEMUSAGE\_LIMIT\_MB'** and **:setting:'MEMUSAGE\_WARNING\_MB'**, at fixed time intervals.

This sets the length of these intervals, in seconds.

See *Memory usage extension*.

## MEMUSAGE\_NOTIFY\_MAIL

Default: False

Scope: scrapy.extensions.memusage

A list of emails to notify if the memory limit has been reached.

Example:

```
MEMUSAGE_NOTIFY_MAIL = ['user@example.com']
```

See *Memory usage extension*.

## MEMUSAGE\_WARNING\_MB

Default: 0

Scope: scrapy.extensions.memusage

The maximum amount of memory to allow (in megabytes) before sending a warning email notifying about it. If zero, no warning will be produced.

## NEWSPIDER\_MODULE

Default: ''

Module where to create new spiders using the **genspider** command.

Example:

```
NEWSPIDER_MODULE = 'mybot.spiders_dev'
```

## RANDOMIZE\_DOWNLOAD\_DELAY

Default: True

If enabled, Scrapy will wait a random amount of time (between  $0.5 * \text{:setting:}\text{'DOWNLOAD\_DELAY'}$  and  $1.5 * \text{:setting:}\text{'DOWNLOAD\_DELAY'}$ ) while fetching requests from the same website.

This randomization decreases the chance of the crawler being detected (and subsequently blocked) by sites which analyze requests looking for statistically significant similarities in the time between their requests.

The randomization policy is the same used by `wget --random-wait` option.

If **:setting:‘DOWNLOAD\_DELAY‘** is zero (default) this option has no effect.

## REACTOR\_THREADPOOL\_MAXSIZE

Default: 10

The maximum limit for Twisted Reactor thread pool size. This is common multi-purpose thread pool used by various Scrapy components. Threaded DNS Resolver, BlockingFeedStorage, S3FilesStore just to name a few. Increase this value if you're experiencing problems with insufficient blocking IO.

## REDIRECT\_MAX\_TIMES

Default: 20

Defines the maximum times a request can be redirected. After this maximum the request's response is returned as is. We used Firefox default value for the same task.

## REDIRECT\_PRIORITY\_ADJUST

Default: +2

Scope: `scrapy.downloadermiddlewares.redirect.RedirectMiddleware`

Adjust redirect request priority relative to original request:

- a positive priority adjust (default) means higher priority.
- a negative priority adjust means lower priority.

## RETRY\_PRIORITY\_ADJUST

Default: -1

Scope: `scrapy.downloadermiddlewares.retry.RetryMiddleware`

Adjust retry request priority relative to original request:

- a positive priority adjust means higher priority.
- **a negative priority adjust (default) means lower priority.**

## ROBOTSTXT\_OBEY

Default: False

Scope: `scrapy.downloadermiddlewares.robotstxt`

If enabled, Scrapy will respect robots.txt policies. For more information see [\*RobotsTxtMiddleware\*](#).

---

**Note:** While the default value is False for historical reasons, this option is enabled by default in settings.py file generated by `scrapy startproject` command.

---

## SCHEDULER

Default: `'scrapy.core.scheduler.Scheduler'`

The scheduler to use for crawling.

## SCHEDULER\_DEBUG

Default: False

Setting to True will log debug information about the requests scheduler. This currently logs (only once) if the requests cannot be serialized to disk. Stats counter (`scheduler/unserializable`) tracks the number of times this happens.

Example entry in logs:

```
1956-01-31 00:00:00+0800 [scrapy.core.scheduler] ERROR: Unable to serialize request:
<GET http://example.com> - reason: cannot serialize <Request at 0x9a7c7ec>
(type Request)> - no more unserializable requests will be logged
(see 'scheduler/unserializable' stats counter)
```

## SCHEDULER\_DISK\_QUEUE

Default: `'scrapy.squeues.PickleLifoDiskQueue'`

Type of disk queue that will be used by scheduler. Other available types are `scrapy.squeues.PickleFifoDiskQueue`, `scrapy.squeues.MarshalFifoDiskQueue`, `scrapy.squeues.MarshalLifoDiskQueue`.



## SCHEDULER\_MEMORY\_QUEUE

Default: `'scrapy.squeues.LifoMemoryQueue'`

Type of in-memory queue used by scheduler. Other available type is: `scrapy.squeues.FifoMemoryQueue`.

## SCHEDULER\_PRIORITY\_QUEUE

Default: `'queuelib.PriorityQueue'`

Type of priority queue used by scheduler.

## SPIDER\_CONTRACTS

Default:: `{}`

A dict containing the spider contracts enabled in your project, used for testing spiders. For more info see [Spiders Contracts](#).

## SPIDER\_CONTRACTS\_BASE

Default:

```
{
    'scrapy.contracts.default.UrlContract' : 1,
    'scrapy.contracts.default.ReturnsContract': 2,
    'scrapy.contracts.default.ScrapesContract': 3,
}
```

A dict containing the scrapy contracts enabled by default in Scrapy. You should never modify this setting in your project, modify **:setting:‘SPIDER\_CONTRACTS’** instead. For more info see [Spiders Contracts](#).

You can disable any of these contracts by assigning `None` to their class path in **:setting:‘SPIDER\_CONTRACTS’**. E.g., to disable the built-in `ScrapesContract`, place this in your `settings.py`:

```
SPIDER_CONTRACTS = {
    'scrapy.contracts.default.ScrapesContract': None,
}
```

## SPIDER\_LOADER\_CLASS

Default: `'scrapy.spiderloader.SpiderLoader'`

The class that will be used for loading spiders, which must implement the [SpiderLoader API](#).

## SPIDER\_LOADER\_WARN\_ONLY

New in version 1.3.3.

Default: `False`

By default, when scrapy tries to import spider classes from **:setting:‘SPIDER\_MODULES’**, it will fail loudly if there is any `ImportError` exception. But you can choose to silence this exception and turn it into a simple warning by setting `SPIDER_LOADER_WARN_ONLY = True`.

**Note:** Some *scrapy commands* run with this setting to `True` already (i.e. they will only issue a warning and will not fail) since they do not actually need to load spider classes to work: **scrapy runspider** <runspider>, **scrapy settings** <settings>, **scrapy startproject** <startproject>, **scrapy version** <version>.

---

## SPIDER\_MIDDLEWARES

Default:: {}

A dict containing the spider middlewares enabled in your project, and their orders. For more info see *Activating a spider middleware*.

## SPIDER\_MIDDLEWARES\_BASE

Default:

```
{
    'scrapy.spidermiddlewares.httperror.HttpErrorMiddleware': 50,
    'scrapy.spidermiddlewares.offsite.OffsiteMiddleware': 500,
    'scrapy.spidermiddlewares.referer.RefererMiddleware': 700,
    'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware': 800,
    'scrapy.spidermiddlewares.depth.DepthMiddleware': 900,
}
```

A dict containing the spider middlewares enabled by default in Scrapy, and their orders. Low orders are closer to the engine, high orders are closer to the spider. For more info see *Activating a spider middleware*.

## SPIDER\_MODULES

Default: []

A list of modules where Scrapy will look for spiders.

Example:

```
SPIDER_MODULES = ['mybot.spiders_prod', 'mybot.spiders_dev']
```

## STATS\_CLASS

Default: 'scrapy.statscollectors.MemoryStatsCollector'

The class to use for collecting stats, who must implement the *Stats Collector API*.

## STATS\_DUMP

Default: True

Dump the *Scrapy stats* (to the Scrapy log) once the spider finishes.

For more info see: *Stats Collection*.

## STATSMAILER\_RCPTS

Default: [] (empty list)

Send Scrapy stats after spiders finish scraping. See `StatsMailer` for more info.

## TELNETCONSOLE\_ENABLED

Default: `True`

A boolean which specifies if the *telnet console* will be enabled (provided its extension is also enabled).

## TELNETCONSOLE\_PORT

Default: [6023, 6073]

The port range to use for the telnet console. If set to `None` or `0`, a dynamically assigned port is used. For more info see *Telnet Console*.

## TEMPLATES\_DIR

Default: `templates` dir inside scrapy module

The directory where to look for templates when creating new projects with **startproject** command and new spiders with **genspider** command.

The project name must not conflict with the name of custom files or directories in the `project` subdirectory.

## URLLENGTH\_LIMIT

Default: 2083

Scope: `spidermiddlewares.urllength`

The maximum URL length to allow for crawled URLs. For more information about the default value for this setting see: <https://boutell.com/newfaq/misc/urllength.html>

## USER\_AGENT

Default: `"Scrapy/VERSION (+https://scrapy.org) "`

The default User-Agent to use when crawling, unless overridden.

### Settings documented elsewhere:

The following settings are documented elsewhere, please check each specific case to see how to enable and use them.

## 3.12 Exceptions

### 3.12.1 Built-in Exceptions reference

Here's a list of all exceptions included in Scrapy and their usage.

## DropItem

**exception** scrapy.exceptions.DropItem

The exception that must be raised by item pipeline stages to stop processing an Item. For more information see [Item Pipeline](#).

## CloseSpider

**exception** scrapy.exceptions.CloseSpider(*reason='cancelled'*)

This exception can be raised from a spider callback to request the spider to be closed/stopped. Supported arguments:

**Parameters** *reason* (*str*) – the reason for closing

For example:

```
def parse_page(self, response):
    if 'Bandwidth exceeded' in response.body:
        raise CloseSpider('bandwidth_exceeded')
```

## DontCloseSpider

**exception** scrapy.exceptions.DontCloseSpider

This exception can be raised in a **:signal:'spider\_idle'** signal handler to prevent the spider from being closed.

## IgnoreRequest

**exception** scrapy.exceptions.IgnoreRequest

This exception can be raised by the Scheduler or any downloader middleware to indicate that the request should be ignored.

## NotConfigured

**exception** scrapy.exceptions.NotConfigured

This exception can be raised by some components to indicate that they will remain disabled. Those components include:

- Extensions
- Item pipelines
- Downloader middlewares
- Spider middlewares

The exception must be raised in the component's `__init__` method.

## NotSupported

**exception** scrapy.exceptions.**NotSupported**

This exception is raised to indicate an unsupported feature.

*Command line tool* Learn about the command-line tool used to manage your Scrapy project.

*Spiders* Write the rules to crawl your websites.

*Selectors* Extract the data from web pages using XPath.

*Scrapy shell* Test your extraction code in an interactive environment.

*Items* Define the data you want to scrape.

*Item Loaders* Populate your items with the extracted data.

*Item Pipeline* Post-process and store your scraped data.

*Feed exports* Output your scraped data using different formats and storages.

*Requests and Responses* Understand the classes used to represent HTTP requests and responses.

*Link Extractors* Convenient classes to extract links to follow from pages.

*Settings* Learn how to configure Scrapy and see all *available settings*.

*Exceptions* See all available exceptions and their meaning.



### 4.1 Logging

---

**Note:** `scrapy.log` has been deprecated alongside its functions in favor of explicit calls to the Python standard logging. Keep reading to learn more about the new logging system.

---

Scrapy uses [Python's builtin logging system](#) for event logging. We'll provide some simple examples to get you started, but for more advanced use-cases it's strongly suggested to read thoroughly its documentation.

Logging works out of the box, and can be configured to some extent with the Scrapy settings listed in [Logging settings](#).

Scrapy calls `scrapy.utils.log.configure_logging()` to set some reasonable defaults and handle those settings in [Logging settings](#) when running commands, so it's recommended to manually call it if you're running Scrapy from scripts as described in [Run Scrapy from a script](#).

#### 4.1.1 Log levels

Python's builtin logging defines 5 different levels to indicate the severity of a given log message. Here are the standard ones, listed in decreasing order:

1. `logging.CRITICAL` - for critical errors (highest severity)
2. `logging.ERROR` - for regular errors
3. `logging.WARNING` - for warning messages
4. `logging.INFO` - for informational messages
5. `logging.DEBUG` - for debugging messages (lowest severity)

#### 4.1.2 How to log messages

Here's a quick example of how to log a message using the `logging.WARNING` level:

```
import logging
logging.warning("This is a warning")
```

There are shortcuts for issuing log messages on any of the standard 5 levels, and there’s also a general `logging.log` method which takes a given level as argument. If needed, the last example could be rewritten as:

```
import logging
logging.log(logging.WARNING, "This is a warning")
```

On top of that, you can create different “loggers” to encapsulate messages. (For example, a common practice is to create different loggers for every module). These loggers can be configured independently, and they allow hierarchical constructions.

The previous examples use the root logger behind the scenes, which is a top level logger where all messages are propagated to (unless otherwise specified). Using `logging` helpers is merely a shortcut for getting the root logger explicitly, so this is also an equivalent of the last snippets:

```
import logging
logger = logging.getLogger()
logger.warning("This is a warning")
```

You can use a different logger just by getting its name with the `logging.getLogger` function:

```
import logging
logger = logging.getLogger('mycustomlogger')
logger.warning("This is a warning")
```

Finally, you can ensure having a custom logger for any module you’re working on by using the `__name__` variable, which is populated with current module’s path:

```
import logging
logger = logging.getLogger(__name__)
logger.warning("This is a warning")
```

**See also:**

**Module `logging`**, **HowTo** Basic Logging Tutorial

**Module `logging`**, **Loggers** Further documentation on loggers

### 4.1.3 Logging from Spiders

Scrapy provides a *logger* within each Spider instance, which can be accessed and used like this:

```
import scrapy

class MySpider(scrapy.Spider):

    name = 'myspider'
    start_urls = ['https://scrapinghub.com']

    def parse(self, response):
        self.logger.info('Parse function called on %s', response.url)
```

That logger is created using the Spider’s name, but you can use any custom Python logger you want. For example:



```
import logging
import scrapy

logger = logging.getLogger('mycustomlogger')

class MySpider(scrapy.Spider):

    name = 'myspider'
    start_urls = ['https://scrapinghub.com']

    def parse(self, response):
        logger.info('Parse function called on %s', response.url)
```

#### 4.1.4 Logging configuration

Loggers on their own don't manage how messages sent through them are displayed. For this task, different “handlers” can be attached to any logger instance and they will redirect those messages to appropriate destinations, such as the standard output, files, emails, etc.

By default, Scrapy sets and configures a handler for the root logger, based on the settings below.

##### Logging settings

These settings can be used to configure the logging:

- **:setting:'LOG\_FILE'**
- **:setting:'LOG\_ENABLED'**
- **:setting:'LOG\_ENCODING'**
- **:setting:'LOG\_LEVEL'**
- **:setting:'LOG\_FORMAT'**
- **:setting:'LOG\_DATEFORMAT'**
- **:setting:'LOG\_STDOUT'**
- **:setting:'LOG\_SHORT\_NAMES'**

The first couple of settings define a destination for log messages. If **:setting:'LOG\_FILE'** is set, messages sent through the root logger will be redirected to a file named **:setting:'LOG\_FILE'** with encoding **:setting:'LOG\_ENCODING'**. If unset and **:setting:'LOG\_ENABLED'** is `True`, log messages will be displayed on the standard error. Lastly, if **:setting:'LOG\_ENABLED'** is `False`, there won't be any visible log output.

**:setting:'LOG\_LEVEL'** determines the minimum level of severity to display, those messages with lower severity will be filtered out. It ranges through the possible levels listed in [Log levels](#).

**:setting:'LOG\_FORMAT'** and **:setting:'LOG\_DATEFORMAT'** specify formatting strings used as layouts for all messages. Those strings can contain any placeholders listed in [logging's logrecord attributes docs](#) and [datetime's strftime and strptime directives](#) respectively.

If **:setting:'LOG\_SHORT\_NAMES'** is set, then the logs will not display the scrapy component that prints the log. It is unset by default, hence logs contain the scrapy component responsible for that log output.

## Command-line options

There are command-line arguments, available for all commands, that you can use to override some of the Scrapy settings regarding logging.

- `--logfile FILE` Overrides `:setting:'LOG_FILE'`
- `--loglevel/-L LEVEL` Overrides `:setting:'LOG_LEVEL'`
- `--nolog` Sets `:setting:'LOG_ENABLED'` to `False`

See also:

Module `logging.handlers` Further documentation on available handlers

## Advanced customization

Because Scrapy uses `stdlib` logging module, you can customize logging using all features of `stdlib` logging.

For example, let's say you're scraping a website which returns many HTTP 404 and 500 responses, and you want to hide all messages like this:

```
2016-12-16 22:00:06 [scrapy.spidermiddlewares.httperror] INFO: Ignoring
response <500 http://quotes.toscrape.com/page/1-34/>: HTTP status code
is not handled or not allowed
```

The first thing to note is a logger name - it is in brackets: `[scrapy.spidermiddlewares.httperror]`. If you get just `[scrapy]` then `:setting:'LOG_SHORT_NAMES'` is likely set to `True`; set it to `False` and re-run the crawl.

Next, we can see that the message has `INFO` level. To hide it we should set logging level for `scrapy.spidermiddlewares.httperror` higher than `INFO`; next level after `INFO` is `WARNING`. It could be done e.g. in the spider's `__init__` method:

```
import logging
import scrapy

class MySpider(scrapy.Spider):
    # ...
    def __init__(self, *args, **kwargs):
        logger = logging.getLogger('scrapy.spidermiddlewares.httperror')
        logger.setLevel(logging.WARNING)
        super().__init__(*args, **kwargs)
```

If you run this spider again then `INFO` messages from `scrapy.spidermiddlewares.httperror` logger will be gone.

### 4.1.5 scrapy.utils.log module

## 4.2 Stats Collection

Scrapy provides a convenient facility for collecting stats in the form of key/values, where values are often counters. The facility is called the Stats Collector, and can be accessed through the `stats` attribute of the *Crawler API*, as illustrated by the examples in the *Common Stats Collector uses* section below.

However, the Stats Collector is always available, so you can always import it in your module and use its API (to increment or set new stat keys), regardless of whether the stats collection is enabled or not. If it's disabled, the API will still work but it won't collect anything. This is aimed at simplifying the stats collector usage: you should spend no more than one line of code for collecting stats in your spider, Scrapy extension, or whatever code you're using the Stats Collector from.

Another feature of the Stats Collector is that it's very efficient (when enabled) and extremely efficient (almost unnoticeable) when disabled.

The Stats Collector keeps a stats table per open spider which is automatically opened when the spider is opened, and closed when the spider is closed.

### 4.2.1 Common Stats Collector uses

Access the stats collector through the `stats` attribute. Here is an example of an extension that access stats:

```
class ExtensionThatAccessStats(object):

    def __init__(self, stats):
        self.stats = stats

    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler.stats)
```

Set stat value:

```
stats.set_value('hostname', socket.gethostname())
```

Increment stat value:

```
stats.inc_value('custom_count')
```

Set stat value only if greater than previous:

```
stats.max_value('max_items_scraped', value)
```

Set stat value only if lower than previous:

```
stats.min_value('min_free_memory_percent', value)
```

Get stat value:

```
>>> stats.get_value('custom_count')
1
```

Get all stats:

```
>>> stats.get_stats()
{'custom_count': 1, 'start_time': datetime.datetime(2009, 7, 14, 21, 47, 28, 977139)}
```

### 4.2.2 Available Stats Collectors

Besides the basic `StatsCollector` there are other Stats Collectors available in Scrapy which extend the basic Stats Collector. You can select which Stats Collector to use through the **setting: 'STATS\_CLASS'** setting. The default Stats Collector used is the `MemoryStatsCollector`.

## MemoryStatsCollector

**class** scrapy.statscollectors.MemoryStatsCollector

A simple stats collector that keeps the stats of the last scraping run (for each spider) in memory, after they're closed. The stats can be accessed through the `spider_stats` attribute, which is a dict keyed by spider domain name.

This is the default Stats Collector used in Scrapy.

**spider\_stats**

A dict of dicts (keyed by spider name) containing the stats of the last scraping run for each spider.

## DummyStatsCollector

**class** scrapy.statscollectors.DummyStatsCollector

A Stats collector which does nothing but is very efficient (because it does nothing). This stats collector can be set via the **:setting:'STATS\_CLASS'** setting, to disable stats collect in order to improve performance. However, the performance penalty of stats collection is usually marginal compared to other Scrapy workload like parsing pages.

## 4.3 Sending e-mail

Although Python makes sending e-mails relatively easy via the `smtplib` library, Scrapy provides its own facility for sending e-mails which is very easy to use and it's implemented using `Twisted non-blocking IO`, to avoid interfering with the non-blocking IO of the crawler. It also provides a simple API for sending attachments and it's very easy to configure, with a few *settings*.

### 4.3.1 Quick example

There are two ways to instantiate the mail sender. You can instantiate it using the standard constructor:

```
from scrapy.mail import MailSender
mailer = MailSender()
```

Or you can instantiate it passing a Scrapy settings object, which will respect the *settings*:

```
mailer = MailSender.from_settings(settings)
```

And here is how to use it to send an e-mail (without attachments):

```
mailer.send(to=["someone@example.com"], subject="Some subject", body="Some body", cc=[
    ↪ "another@example.com"])
```

### 4.3.2 MailSender class reference

MailSender is the preferred class to use for sending emails from Scrapy, as it uses `Twisted non-blocking IO`, like the rest of the framework.

**class** scrapy.mail.MailSender(*smtphost=None, mailfrom=None, smtpuser=None, smtp-pass=None, smtpport=None*)

**Parameters**

- **smtphost** (*str or bytes*) – the SMTP host to use for sending the emails. If omitted, the **:setting:‘MAIL\_HOST’** setting will be used.
- **mailfrom** (*str*) – the address used to send emails (in the `From:` header). If omitted, the **:setting:‘MAIL\_FROM’** setting will be used.
- **smtpuser** – the SMTP user. If omitted, the **:setting:‘MAIL\_USER’** setting will be used. If not given, no SMTP authentication will be performed.
- **smtppass** (*str or bytes*) – the SMTP pass for authentication.
- **smtpport** (*int*) – the SMTP port to connect to
- **smtptls** (*boolean*) – enforce using SMTP STARTTLS
- **smtppsll** (*boolean*) – enforce using a secure SSL connection

**classmethod from\_settings** (*settings*)

Instantiate using a Scrapy settings object, which will respect *these Scrapy settings*.

**Parameters** **settings** (scrapy.settings.Settings object) – the e-mail recipients

**send** (*to, subject, body, cc=None, attachs=(), mimetype='text/plain', charset=None*)

Send email to the given recipients.

#### Parameters

- **to** (*str or list of str*) – the e-mail recipients
- **subject** (*str*) – the subject of the e-mail
- **cc** (*str or list of str*) – the e-mails to CC
- **body** (*str*) – the e-mail body
- **attachs** (*iterable*) – an iterable of tuples (*attach\_name, mimetype, file\_object*) where *attach\_name* is a string with the name that will appear on the e-mail's attachment, *mimetype* is the mimetype of the attachment and *file\_object* is a readable file object with the contents of the attachment
- **mimetype** (*str*) – the MIME type of the e-mail
- **charset** (*str*) – the character encoding to use for the e-mail contents

### 4.3.3 Mail settings

These settings define the default constructor values of the *MailSender* class, and can be used to configure e-mail notifications in your project without writing any code (for those extensions and code that uses *MailSender*).

#### MAIL\_FROM

Default: 'scrapy@localhost'

Sender email to use (`From:` header) for sending emails.

#### MAIL\_HOST

Default: 'localhost'

SMTP host to use for sending emails.

## MAIL\_PORT

Default: 25

SMTP port to use for sending emails.

## MAIL\_USER

Default: None

User to use for SMTP authentication. If disabled no SMTP authentication will be performed.

## MAIL\_PASS

Default: None

Password to use for SMTP authentication, along with **:setting:‘MAIL\_USER‘**.

## MAIL\_TLS

Default: False

Enforce using STARTTLS. STARTTLS is a way to take an existing insecure connection, and upgrade it to a secure connection using SSL/TLS.

## MAIL\_SSL

Default: False

Enforce connecting using an SSL encrypted connection

## 4.4 Telnet Console

Scrapy comes with a built-in telnet console for inspecting and controlling a Scrapy running process. The telnet console is just a regular python shell running inside the Scrapy process, so you can do literally anything from it.

The telnet console is a *built-in Scrapy extension* which comes enabled by default, but you can also disable it if you want. For more information about the extension itself see *Telnet console extension*.

### 4.4.1 How to access the telnet console

The telnet console listens in the TCP port defined in the **:setting:‘TELNETCONSOLE\_PORT‘** setting, which defaults to 6023. To access the console you need to type:

```
telnet localhost 6023
>>>
```

You need the telnet program which comes installed by default in Windows, and most Linux distros.

### 4.4.2 Available variables in the telnet console

The telnet console is like a regular Python shell running inside the Scrapy process, so you can do anything from it including importing new modules, etc.

However, the telnet console comes with some default variables defined for convenience:

Shortcut	Description
<code>crawler</code>	the Scrapy Crawler ( <i>scrapy.crawler.Crawler</i> object)
<code>engine</code>	Crawler.engine attribute
<code>spider</code>	the active spider
<code>slot</code>	the engine slot
<code>extensions</code>	the Extension Manager (Crawler.extensions attribute)
<code>stats</code>	the Stats Collector (Crawler.stats attribute)
<code>settings</code>	the Scrapy settings object (Crawler.settings attribute)
<code>est</code>	print a report of the engine status
<code>prefs</code>	for memory debugging (see <i>Debugging memory leaks</i> )
<code>p</code>	a shortcut to the <code>pprint.pprint</code> function
<code>hpy</code>	for memory debugging (see <i>Debugging memory leaks</i> )

### 4.4.3 Telnet console usage examples

Here are some example tasks you can do with the telnet console:

#### View engine status

You can use the `est()` method of the Scrapy engine to quickly show its state using the telnet console:

```
telnet localhost 6023
>>> est()
Execution engine status

time()-engine.start_time           : 8.62972998619
engine.has_capacity()               : False
len(engine.downloader.active)       : 16
engine.scrapers.is_idle()           : False
engine.spider.name                  : followall
engine.spider_is_idle(engine.spider): False
engine.slot.closing                 : False
len(engine.slot.inprogress)         : 16
len(engine.slot.scheduler.dqs or []) : 0
len(engine.slot.scheduler.mqs)      : 92
len(engine.scrapers.slot.queue)      : 0
len(engine.scrapers.slot.active)     : 0
engine.scrapers.slot.active_size     : 0
engine.scrapers.slot.itemproc_size   : 0
engine.scrapers.slot.needs_backout() : False
```

#### Pause, resume and stop the Scrapy engine

To pause:

```
telnet localhost 6023
>>> engine.pause()
>>>
```

To resume:

```
telnet localhost 6023
>>> engine.unpause()
>>>
```

To stop:

```
telnet localhost 6023
>>> engine.stop()
Connection closed by foreign host.
```

## 4.4.4 Telnet Console signals

`scrapy.extensions.telnet.update_telnet_vars` (*telnet\_vars*)

Sent just before the telnet console is opened. You can hook up to this signal to add, remove or update the variables that will be available in the telnet local namespace. In order to do that, you need to update the `telnet_vars` dict in your handler.

**Parameters** `telnet_vars` (*dict*) – the dict of telnet variables

## 4.4.5 Telnet settings

These are the settings that control the telnet console's behaviour:

### TELNETCONSOLE\_PORT

Default: `[6023, 6073]`

The port range to use for the telnet console. If set to `None` or `0`, a dynamically assigned port is used.

### TELNETCONSOLE\_HOST

Default: `'127.0.0.1'`

The interface the telnet console should listen on

## 4.5 Web Service

webservice has been moved into a separate project.

It is hosted at:

<https://github.com/scrapy-plugins/scrapy-jsonrpc>

**Logging** Learn how to use Python's builtin logging on Scrapy.

**Stats Collection** Collect statistics about your scraping crawler.



*Sending e-mail* Send email notifications when certain events occur.

*Telnet Console* Inspect a running crawler using a built-in Python console.

*Web Service* Monitor and control a crawler using a web service.



---

## Solving specific problems

---

### 5.1 Frequently Asked Questions

#### 5.1.1 How does Scrapy compare to BeautifulSoup or lxml?

`BeautifulSoup` and `lxml` are libraries for parsing HTML and XML. Scrapy is an application framework for writing web spiders that crawl web sites and extract data from them.

Scrapy provides a built-in mechanism for extracting data (called *selectors*) but you can easily use `BeautifulSoup` (or `lxml`) instead, if you feel more comfortable working with them. After all, they're just parsing libraries which can be imported and used from any Python code.

In other words, comparing `BeautifulSoup` (or `lxml`) to Scrapy is like comparing `jinja2` to `Django`.

#### 5.1.2 Can I use Scrapy with BeautifulSoup?

Yes, you can. As mentioned *above*, `BeautifulSoup` can be used for parsing HTML responses in Scrapy callbacks. You just have to feed the response's body into a `BeautifulSoup` object and extract whatever data you need from it.

Here's an example spider using BeautifulSoup API, with `lxml` as the HTML parser:

```
from bs4 import BeautifulSoup
import scrapy

class ExampleSpider(scrapy.Spider):
    name = "example"
    allowed_domains = ["example.com"]
    start_urls = (
        'http://www.example.com/',
    )

    def parse(self, response):
```

(continues on next page)

(continued from previous page)

```
# use lxml to get decent HTML parsing speed
soup = BeautifulSoup(response.text, 'lxml')
yield {
    "url": response.url,
    "title": soup.h1.string
}
```

---

**Note:** BeautifulSoup supports several HTML/XML parsers. See [BeautifulSoup’s official documentation](#) on which ones are available.

---

### 5.1.3 What Python versions does Scrapy support?

Scrapy is supported under Python 2.7 and Python 3.4+ under CPython (default Python implementation) and PyPy (starting with PyPy 5.9). Python 2.6 support was dropped starting at Scrapy 0.20. Python 3 support was added in Scrapy 1.1. PyPy support was added in Scrapy 1.4, PyPy3 support was added in Scrapy 1.5.

---

**Note:** For Python 3 support on Windows, it is recommended to use Anaconda/Miniconda as *outlined in the installation guide*.

---

### 5.1.4 Did Scrapy “steal” X from Django?

Probably, but we don’t like that word. We think [Django](#) is a great open source project and an example to follow, so we’ve used it as an inspiration for Scrapy.

We believe that, if something is already done well, there’s no need to reinvent it. This concept, besides being one of the foundations for open source and free software, not only applies to software but also to documentation, procedures, policies, etc. So, instead of going through each problem ourselves, we choose to copy ideas from those projects that have already solved them properly, and focus on the real problems we need to solve.

We’d be proud if Scrapy serves as an inspiration for other projects. Feel free to steal from us!

### 5.1.5 Does Scrapy work with HTTP proxies?

Yes. Support for HTTP proxies is provided (since Scrapy 0.8) through the HTTP Proxy downloader middleware. See [HttpProxyMiddleware](#).

### 5.1.6 How can I scrape an item with attributes in different pages?

See [Passing additional data to callback functions](#).

### 5.1.7 Scrapy crashes with: ImportError: No module named win32api

You need to install [pywin32](#) because of [this Twisted bug](#).

### 5.1.8 How can I simulate a user login in my spider?

See *Using FormRequest.from\_response() to simulate a user login*.

### 5.1.9 Does Scrapy crawl in breadth-first or depth-first order?

By default, Scrapy uses a **LIFO** queue for storing pending requests, which basically means that it crawls in **DFO** order. This order is more convenient in most cases. If you do want to crawl in true **BFO** order, you can do it by setting the following settings:

```
DEPTH_PRIORITY = 1
SCHEDULER_DISK_QUEUE = 'scrapy.squeues.PickleFifoDiskQueue'
SCHEDULER_MEMORY_QUEUE = 'scrapy.squeues.FifoMemoryQueue'
```

### 5.1.10 My Scrapy crawler has memory leaks. What can I do?

See *Debugging memory leaks*.

Also, Python has a builtin memory leak issue which is described in *Leaks without leaks*.

### 5.1.11 How can I make Scrapy consume less memory?

See previous question.

### 5.1.12 Can I use Basic HTTP Authentication in my spiders?

Yes, see *HttpAuthMiddleware*.

### 5.1.13 Why does Scrapy download pages in English instead of my native language?

Try changing the default `Accept-Language` request header by overriding the **setting: 'DEFAULT\_REQUEST\_HEADERS'** setting.

### 5.1.14 Where can I find some example Scrapy projects?

See *Examples*.

### 5.1.15 Can I run a spider without creating a project?

Yes. You can use the **runspider** command. For example, if you have a spider written in a `my_spider.py` file you can run it with:

```
scrapy runspider my_spider.py
```

See **runspider** command for more info.

### 5.1.16 I get “Filtered offsite request” messages. How can I fix them?

Those messages (logged with `DEBUG` level) don’t necessarily mean there is a problem, so you may not need to fix them.

Those messages are thrown by the Offsite Spider Middleware, which is a spider middleware (enabled by default) whose purpose is to filter out requests to domains outside the ones covered by the spider.

For more info see: *OffsiteMiddleware*.

### 5.1.17 What is the recommended way to deploy a Scrapy crawler in production?

See *Deploying Spiders*.

### 5.1.18 Can I use JSON for large exports?

It’ll depend on how large your output is. See *this warning* in *JsonItemExporter* documentation.

### 5.1.19 Can I return (Twisted) deferreds from signal handlers?

Some signals support returning deferreds from their handlers, others don’t. See the *Built-in signals reference* to know which ones.

### 5.1.20 What does the response status code 999 means?

999 is a custom response status code used by Yahoo sites to throttle requests. Try slowing down the crawling speed by using a download delay of 2 (or higher) in your spider:

```
class MySpider(CrawlSpider):  
    name = 'myspider'  
  
    download_delay = 2  
  
    # [ ... rest of the spider code ... ]
```

Or by setting a global download delay in your project with the **:setting:‘DOWNLOAD\_DELAY’** setting.

### 5.1.21 Can I call `pdb.set_trace()` from my spiders to debug them?

Yes, but you can also use the Scrapy shell which allows you to quickly analyze (and even modify) the response being processed by your spider, which is, quite often, more useful than plain old `pdb.set_trace()`.

For more info see *Invoking the shell from spiders to inspect responses*.

### 5.1.22 Simplest way to dump all my scraped items into a JSON/CSV/XML file?

To dump into a JSON file:

```
scrapy crawl myspider -o items.json
```

To dump into a CSV file:

```
scrapy crawl myspider -o items.csv
```

To dump into a XML file:

```
scrapy crawl myspider -o items.xml
```

For more information see *Feed exports*

### 5.1.23 What's this huge cryptic `__VIEWSTATE` parameter used in some forms?

The `__VIEWSTATE` parameter is used in sites built with ASP.NET/VB.NET. For more info on how it works see [this page](#). Also, here's an [example spider](#) which scrapes one of these sites.

### 5.1.24 What's the best way to parse big XML/CSV data feeds?

Parsing big feeds with XPath selectors can be problematic since they need to build the DOM of the entire feed in memory, and this can be quite slow and consume a lot of memory.

In order to avoid parsing all the entire feed at once in memory, you can use the functions `xmliter` and `csviter` from `scrapy.utils.iterators` module. In fact, this is what the feed spiders (see *Spiders*) use under the cover.

### 5.1.25 Does Scrapy manage cookies automatically?

Yes, Scrapy receives and keeps track of cookies sent by servers, and sends them back on subsequent requests, like any regular web browser does.

For more info see *Requests and Responses* and *CookiesMiddleware*.

### 5.1.26 How can I see the cookies being sent and received from Scrapy?

Enable the `:setting:'COOKIES_DEBUG'` setting.

### 5.1.27 How can I instruct a spider to stop itself?

Raise the `CloseSpider` exception from a callback. For more info see: *CloseSpider*.

### 5.1.28 How can I prevent my Scrapy bot from getting banned?

See *Avoiding getting banned*.

### 5.1.29 Should I use spider arguments or settings to configure my spider?

Both *spider arguments* and *settings* can be used to configure your spider. There is no strict rule that mandates to use one or the other, but settings are more suited for parameters that, once set, don't change much, while spider arguments are meant to change more often, even on each spider run and sometimes are required for the spider to run at all (for example, to set the start url of a spider).

To illustrate with an example, assuming you have a spider that needs to log into a site to scrape data, and you only want to scrape data from a certain section of the site (which varies each time). In that case, the credentials to log in would be settings, while the url of the section to scrape would be a spider argument.

### 5.1.30 I'm scraping a XML document and my XPath selector doesn't return any items

You may need to remove namespaces. See [Removing namespaces](#).

## 5.2 Debugging Spiders

This document explains the most common techniques for debugging spiders. Consider the following scrapy spider below:

```
import scrapy
from myproject.items import MyItem

class MySpider(scrapy.Spider):
    name = 'myspider'
    start_urls = (
        'http://example.com/page1',
        'http://example.com/page2',
    )

    def parse(self, response):
        # collect `item_urls`
        for item_url in item_urls:
            yield scrapy.Request(item_url, self.parse_item)

    def parse_item(self, response):
        item = MyItem()
        # populate `item` fields
        # and extract item_details_url
        yield scrapy.Request(item_details_url, self.parse_details, meta={'item': item})
    ↪)

    def parse_details(self, response):
        item = response.meta['item']
        # populate more `item` fields
        return item
```

Basically this is a simple spider which parses two pages of items (the start\_urls). Items also have a details page with additional information, so we use the meta functionality of [Request](#) to pass a partially populated item.

### 5.2.1 Parse Command

The most basic way of checking the output of your spider is to use the **parse** command. It allows to check the behaviour of different parts of the spider at the method level. It has the advantage of being flexible and simple to use, but does not allow debugging code inside a method.

In order to see the item scraped from a specific url:



```
$ scrapy parse --spider=mypider -c parse_item -d 2 <item_url>
[ ... scrapy log lines crawling example.com spider ... ]

>>> STATUS DEPTH LEVEL 2 <<<
# Scraped Items -----
[{'url': <item_url>}]

# Requests -----
[]
```

Using the `--verbose` or `-v` option we can see the status at each depth level:

```
$ scrapy parse --spider=mypider -c parse_item -d 2 -v <item_url>
[ ... scrapy log lines crawling example.com spider ... ]

>>> DEPTH LEVEL: 1 <<<
# Scraped Items -----
[]

# Requests -----
[<GET item_details_url>]

>>> DEPTH LEVEL: 2 <<<
# Scraped Items -----
[{'url': <item_url>}]

# Requests -----
[]
```

Checking items scraped from a single start\_url, can also be easily achieved using:

```
$ scrapy parse --spider=mypider -d 3 'http://example.com/page1'
```

## 5.2.2 Scrapy Shell

While the **parse** command is very useful for checking behaviour of a spider, it is of little help to check what happens inside a callback, besides showing the response received and the output. How to debug the situation when `parse_details` sometimes receives no item?

Fortunately, the **shell** is your bread and butter in this case (see *Invoking the shell from spiders to inspect responses*):

```
from scrapy.shell import inspect_response

def parse_details(self, response):
    item = response.meta.get('item', None)
    if item:
        # populate more `item` fields
        return item
    else:
        inspect_response(response, self)
```

See also: *Invoking the shell from spiders to inspect responses*.

### 5.2.3 Open in browser

Sometimes you just want to see how a certain response looks in a browser, you can use the `open_in_browser` function for that. Here is an example of how you would use it:

```
from scrapy.utils.response import open_in_browser

def parse_details(self, response):
    if "item name" not in response.body:
        open_in_browser(response)
```

`open_in_browser` will open a browser with the response received by Scrapy at that point, adjusting the `base` tag so that images and styles are displayed properly.

### 5.2.4 Logging

Logging is another useful option for getting information about your spider run. Although not as convenient, it comes with the advantage that the logs will be available in all future runs should they be necessary again:

```
def parse_details(self, response):
    item = response.meta.get('item', None)
    if item:
        # populate more `item` fields
        return item
    else:
        self.logger.warning('No item received for %s', response.url)
```

For more information, check the [Logging](#) section.

## 5.3 Spiders Contracts

New in version 0.15.

---

**Note:** This is a new feature (introduced in Scrapy 0.15) and may be subject to minor functionality/API updates. Check the release notes to be notified of updates.

---

Testing spiders can get particularly annoying and while nothing prevents you from writing unit tests the task gets cumbersome quickly. Scrapy offers an integrated way of testing your spiders by the means of contracts.

This allows you to test each callback of your spider by hardcoding a sample url and check various constraints for how the callback processes the response. Each contract is prefixed with an `@` and included in the docstring. See the following example:

```
def parse(self, response):
    """ This function parses a sample response. Some contracts are mingled
    with this docstring.

    @url http://www.amazon.com/s?field-keywords=selfish+gene
    @returns items 1 16
    @returns requests 0 0
    @scrapes Title Author Year Price
    """
```

This callback is tested using three built-in contracts:

**class** scrapy.contracts.default.UrlContract

This contract (`@url`) sets the sample url used when checking other contract conditions for this spider. This contract is mandatory. All callbacks lacking this contract are ignored when running the checks:

```
@url url
```

**class** scrapy.contracts.default.ReturnsContract

This contract (`@returns`) sets lower and upper bounds for the items and requests returned by the spider. The upper bound is optional:

```
@returns item(s) | request(s) [min [max]]
```

**class** scrapy.contracts.default.ScrapesContract

This contract (`@scrapes`) checks that all the items returned by the callback have the specified fields:

```
@scrapes field_1 field_2 ...
```

Use the **check** command to run the contract checks.

### 5.3.1 Custom Contracts

If you find you need more power than the built-in scrapy contracts you can create and load your own contracts in the project by using the **setting: 'SPIDER\_CONTRACTS'** setting:

```
SPIDER_CONTRACTS = {
    'myproject.contracts.ResponseCheck': 10,
    'myproject.contracts.ItemValidate': 10,
}
```

Each contract must inherit from `scrapy.contracts.Contract` and can override three methods:

**class** scrapy.contracts.Contract (*method, \*args*)

#### Parameters

- **method** (*function*) – callback function to which the contract is associated
- **args** (*list*) – list of arguments passed into the docstring (whitespace separated)

**adjust\_request\_args** (*args*)

This receives a `dict` as an argument containing default arguments for request object. `Request` is used by default, but this can be changed with the `request_cls` attribute. If multiple contracts in chain have this attribute defined, the last one is used.

Must return the same or a modified version of it.

**pre\_process** (*response*)

This allows hooking in various checks on the response received from the sample request, before it's being passed to the callback.

**post\_process** (*output*)

This allows processing the output of the callback. Iterators are converted listified before being passed to this hook.

Here is a demo contract which checks the presence of a custom header in the response received. Raise `scrapy.exceptions.ContractFail` in order to get the failures pretty printed:

```
from scrapy.contracts import Contract
from scrapy.exceptions import ContractFail

class HasHeaderContract(Contract):
    """ Demo contract which checks the presence of a custom header
        @has_header X-CustomHeader
    """

    name = 'has_header'

    def pre_process(self, response):
        for header in self.args:
            if header not in response.headers:
                raise ContractFail('X-CustomHeader not present')
```

## 5.4 Common Practices

This section documents common practices when using Scrapy. These are things that cover many topics and don't often fall into any other specific section.

### 5.4.1 Run Scrapy from a script

You can use the [API](#) to run Scrapy from a script, instead of the typical way of running Scrapy via `scrapy crawl`.

Remember that Scrapy is built on top of the Twisted asynchronous networking library, so you need to run it inside the Twisted reactor.

The first utility you can use to run your spiders is `scrapy.crawler.CrawlerProcess`. This class will start a Twisted reactor for you, configuring the logging and setting shutdown handlers. This class is the one used by all Scrapy commands.

Here's an example showing how to run a single spider with it.

```
import scrapy
from scrapy.crawler import CrawlerProcess

class MySpider(scrapy.Spider):
    # Your spider definition
    ...

process = CrawlerProcess({
    'USER_AGENT': 'Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)'
})

process.crawl(MySpider)
process.start() # the script will block here until the crawling is finished
```

Make sure to check `CrawlerProcess` documentation to get acquainted with its usage details.

If you are inside a Scrapy project there are some additional helpers you can use to import those components within the project. You can automatically import your spiders passing their name to `CrawlerProcess`, and use `get_project_settings` to get a `Settings` instance with your project settings.

What follows is a working example of how to do that, using the [testspiders](#) project as example.

```

from scrapy.crawler import CrawlerProcess
from scrapy.utils.project import get_project_settings

process = CrawlerProcess(get_project_settings())

# 'followall' is the name of one of the spiders of the project.
process.crawl('followall', domain='scrapinghub.com')
process.start() # the script will block here until the crawling is finished

```

There's another Scrapy utility that provides more control over the crawling process: `scrapy.crawler.CrawlerRunner`. This class is a thin wrapper that encapsulates some simple helpers to run multiple crawlers, but it won't start or interfere with existing reactors in any way.

Using this class the reactor should be explicitly run after scheduling your spiders. It's recommended you use `CrawlerRunner` instead of `CrawlerProcess` if your application is already using Twisted and you want to run Scrapy in the same reactor.

Note that you will also have to shutdown the Twisted reactor yourself after the spider is finished. This can be achieved by adding callbacks to the deferred returned by the `CrawlerRunner.crawl` method.

Here's an example of its usage, along with a callback to manually stop the reactor after *MySpider* has finished running.

```

from twisted.internet import reactor
import scrapy
from scrapy.crawler import CrawlerRunner
from scrapy.utils.log import configure_logging

class MySpider(scrapy.Spider):
    # Your spider definition
    ...

configure_logging({'LOG_FORMAT': '%(levelname)s: %(message)s'})
runner = CrawlerRunner()

d = runner.crawl(MySpider)
d.addBoth(lambda _: reactor.stop())
reactor.run() # the script will block here until the crawling is finished

```

**See also:**

Twisted Reactor Overview.

## 5.4.2 Running multiple spiders in the same process

By default, Scrapy runs a single spider per process when you run `scrapy crawl`. However, Scrapy supports running multiple spiders per process using the *internal API*.

Here is an example that runs multiple spiders simultaneously:

```

import scrapy
from scrapy.crawler import CrawlerProcess

class MySpider1(scrapy.Spider):
    # Your first spider definition
    ...

class MySpider2(scrapy.Spider):

```

(continues on next page)

(continued from previous page)

```
# Your second spider definition
...

process = CrawlerProcess()
process.crawl(MySpider1)
process.crawl(MySpider2)
process.start() # the script will block here until all crawling jobs are finished
```

Same example using CrawlerRunner:

```
import scrapy
from twisted.internet import reactor
from scrapy.crawler import CrawlerRunner
from scrapy.utils.log import configure_logging

class MySpider1(scrapy.Spider):
    # Your first spider definition
    ...

class MySpider2(scrapy.Spider):
    # Your second spider definition
    ...

configure_logging()
runner = CrawlerRunner()
runner.crawl(MySpider1)
runner.crawl(MySpider2)
d = runner.join()
d.addBoth(lambda _: reactor.stop())

reactor.run() # the script will block here until all crawling jobs are finished
```

Same example but running the spiders sequentially by chaining the deferreds:

```
from twisted.internet import reactor, defer
from scrapy.crawler import CrawlerRunner
from scrapy.utils.log import configure_logging

class MySpider1(scrapy.Spider):
    # Your first spider definition
    ...

class MySpider2(scrapy.Spider):
    # Your second spider definition
    ...

configure_logging()
runner = CrawlerRunner()

@defer.inlineCallbacks
def crawl():
    yield runner.crawl(MySpider1)
    yield runner.crawl(MySpider2)
    reactor.stop()

crawl()
reactor.run() # the script will block here until the last crawl call is finished
```

**See also:**

*Run Scrapy from a script.*

### 5.4.3 Distributed crawls

Scrapy doesn't provide any built-in facility for running crawls in a distribute (multi-server) manner. However, there are some ways to distribute crawls, which vary depending on how you plan to distribute them.

If you have many spiders, the obvious way to distribute the load is to setup many Scrapy instances and distribute spider runs among those.

If you instead want to run a single (big) spider through many machines, what you usually do is partition the urls to crawl and send them to each separate spider. Here is a concrete example:

First, you prepare the list of urls to crawl and put them into separate files/urls:

```
http://somedomain.com/urls-to-crawl/spider1/part1.list
http://somedomain.com/urls-to-crawl/spider1/part2.list
http://somedomain.com/urls-to-crawl/spider1/part3.list
```

Then you fire a spider run on 3 different Scrapy servers. The spider would receive a (spider) argument `part` with the number of the partition to crawl:

```
curl http://scrapy1.mycompany.com:6800/schedule.json -d project=myproject -d
↪spider=spider1 -d part=1
curl http://scrapy2.mycompany.com:6800/schedule.json -d project=myproject -d
↪spider=spider1 -d part=2
curl http://scrapy3.mycompany.com:6800/schedule.json -d project=myproject -d
↪spider=spider1 -d part=3
```

### 5.4.4 Avoiding getting banned

Some websites implement certain measures to prevent bots from crawling them, with varying degrees of sophistication. Getting around those measures can be difficult and tricky, and may sometimes require special infrastructure. Please consider contacting [commercial support](#) if in doubt.

Here are some tips to keep in mind when dealing with these kinds of sites:

- rotate your user agent from a pool of well-known ones from browsers (google around to get a list of them)
- disable cookies (see **:setting: 'COOKIES\_ENABLED'**) as some sites may use cookies to spot bot behaviour
- use download delays (2 or higher). See **:setting: 'DOWNLOAD\_DELAY'** setting.
- if possible, use [Google cache](#) to fetch pages, instead of hitting the sites directly
- use a pool of rotating IPs. For example, the free [Tor project](#) or paid services like [ProxyMesh](#). An open source alternative is [scrapoxy](#), a super proxy that you can attach your own proxies to.
- use a highly distributed downloader that circumvents bans internally, so you can just focus on parsing clean pages. One example of such downloaders is [Crawlera](#)

If you are still unable to prevent your bot getting banned, consider contacting [commercial support](#).

## 5.5 Broad Crawls

Scrapy defaults are optimized for crawling specific sites. These sites are often handled by a single Scrapy spider, although this is not necessary or required (for example, there are generic spiders that handle any given site thrown at them).

In addition to this “focused crawl”, there is another common type of crawling which covers a large (potentially unlimited) number of domains, and is only limited by time or other arbitrary constraint, rather than stopping when the domain was crawled to completion or when there are no more requests to perform. These are called “broad crawls” and is the typical crawlers employed by search engines.

These are some common properties often found in broad crawls:

- they crawl many domains (often, unbounded) instead of a specific set of sites
- they don’t necessarily crawl domains to completion, because it would be impractical (or impossible) to do so, and instead limit the crawl by time or number of pages crawled
- they are simpler in logic (as opposed to very complex spiders with many extraction rules) because data is often post-processed in a separate stage
- they crawl many domains concurrently, which allows them to achieve faster crawl speeds by not being limited by any particular site constraint (each site is crawled slowly to respect politeness, but many sites are crawled in parallel)

As said above, Scrapy default settings are optimized for focused crawls, not broad crawls. However, due to its asynchronous architecture, Scrapy is very well suited for performing fast broad crawls. This page summarizes some things you need to keep in mind when using Scrapy for doing broad crawls, along with concrete suggestions of Scrapy settings to tune in order to achieve an efficient broad crawl.

### 5.5.1 Increase concurrency

Concurrency is the number of requests that are processed in parallel. There is a global limit and a per-domain limit.

The default global concurrency limit in Scrapy is not suitable for crawling many different domains in parallel, so you will want to increase it. How much to increase it will depend on how much CPU you crawler will have available. A good starting point is 100, but the best way to find out is by doing some trials and identifying at what concurrency your Scrapy process gets CPU bounded. For optimum performance, you should pick a concurrency where CPU usage is at 80-90%.

To increase the global concurrency use:

```
CONCURRENT_REQUESTS = 100
```

### 5.5.2 Increase Twisted IO thread pool maximum size

Currently Scrapy does DNS resolution in a blocking way with usage of thread pool. With higher concurrency levels the crawling could be slow or even fail hitting DNS resolver timeouts. Possible solution to increase the number of threads handling DNS queries. The DNS queue will be processed faster speeding up establishing of connection and crawling overall.

To increase maximum thread pool size use:

```
REACTOR_THREADPOOL_MAXSIZE = 20
```



### 5.5.3 Setup your own DNS

If you have multiple crawling processes and single central DNS, it can act like DoS attack on the DNS server resulting to slow down of entire network or even blocking your machines. To avoid this setup your own DNS server with local cache and upstream to some large DNS like OpenDNS or Verizon.

### 5.5.4 Reduce log level

When doing broad crawls you are often only interested in the crawl rates you get and any errors found. These stats are reported by Scrapy when using the `INFO` log level. In order to save CPU (and log storage requirements) you should not use `DEBUG` log level when performing large broad crawls in production. Using `DEBUG` level when developing your (broad) crawler may be fine though.

To set the log level use:

```
LOG_LEVEL = 'INFO'
```

### 5.5.5 Disable cookies

Disable cookies unless you *really* need. Cookies are often not needed when doing broad crawls (search engine crawlers ignore them), and they improve performance by saving some CPU cycles and reducing the memory footprint of your Scrapy crawler.

To disable cookies use:

```
COOKIES_ENABLED = False
```

### 5.5.6 Disable retries

Retrying failed HTTP requests can slow down the crawls substantially, specially when sites causes are very slow (or fail) to respond, thus causing a timeout error which gets retried many times, unnecessarily, preventing crawler capacity to be reused for other domains.

To disable retries use:

```
RETRY_ENABLED = False
```

### 5.5.7 Reduce download timeout

Unless you are crawling from a very slow connection (which shouldn't be the case for broad crawls) reduce the download timeout so that stuck requests are discarded quickly and free up capacity to process the next ones.

To reduce the download timeout use:

```
DOWNLOAD_TIMEOUT = 15
```

### 5.5.8 Disable redirects

Consider disabling redirects, unless you are interested in following them. When doing broad crawls it's common to save redirects and resolve them when revisiting the site at a later crawl. This also help to keep the number of request

constant per crawl batch, otherwise redirect loops may cause the crawler to dedicate too many resources on any specific domain.

To disable redirects use:

```
REDIRECT_ENABLED = False
```

## 5.5.9 Enable crawling of “Ajax Crawlable Pages”

Some pages (up to 1%, based on empirical data from year 2013) declare themselves as `ajax crawlable`. This means they provide plain HTML version of content that is usually available only via AJAX. Pages can indicate it in two ways:

1. by using `#!` in URL - this is the default way;
2. by using a special meta tag - this way is used on “main”, “index” website pages.

Scrapy handles (1) automatically; to handle (2) enable *AjaxCrawlMiddleware*:

```
AJAXCRAWL_ENABLED = True
```

When doing broad crawls it’s common to crawl a lot of “index” web pages; *AjaxCrawlMiddleware* helps to crawl them correctly. It is turned OFF by default because it has some performance overhead, and enabling it for focused crawls doesn’t make much sense.

## 5.6 Using your browser’s Developer Tools for scraping

Here is a general guide on how to use your browser’s Developer Tools to ease the scraping process. Today almost all browsers come with built in *Developer Tools* and although we will use Firefox in this guide, the concepts are applicable to any other browser.

In this guide we’ll introduce the basic tools to use from a browser’s Developer Tools by scraping [quotes.toscrape.com](http://quotes.toscrape.com).

### 5.6.1 Caveats with inspecting the live browser DOM

Since Developer Tools operate on a live browser DOM, what you’ll actually see when inspecting the page source is not the original HTML, but a modified one after applying some browser clean up and executing Javascript code. Firefox, in particular, is known for adding `<tbody>` elements to tables. Scrapy, on the other hand, does not modify the original page HTML, so you won’t be able to extract any data if you use `<tbody>` in your XPath expressions.

Therefore, you should keep in mind the following things:

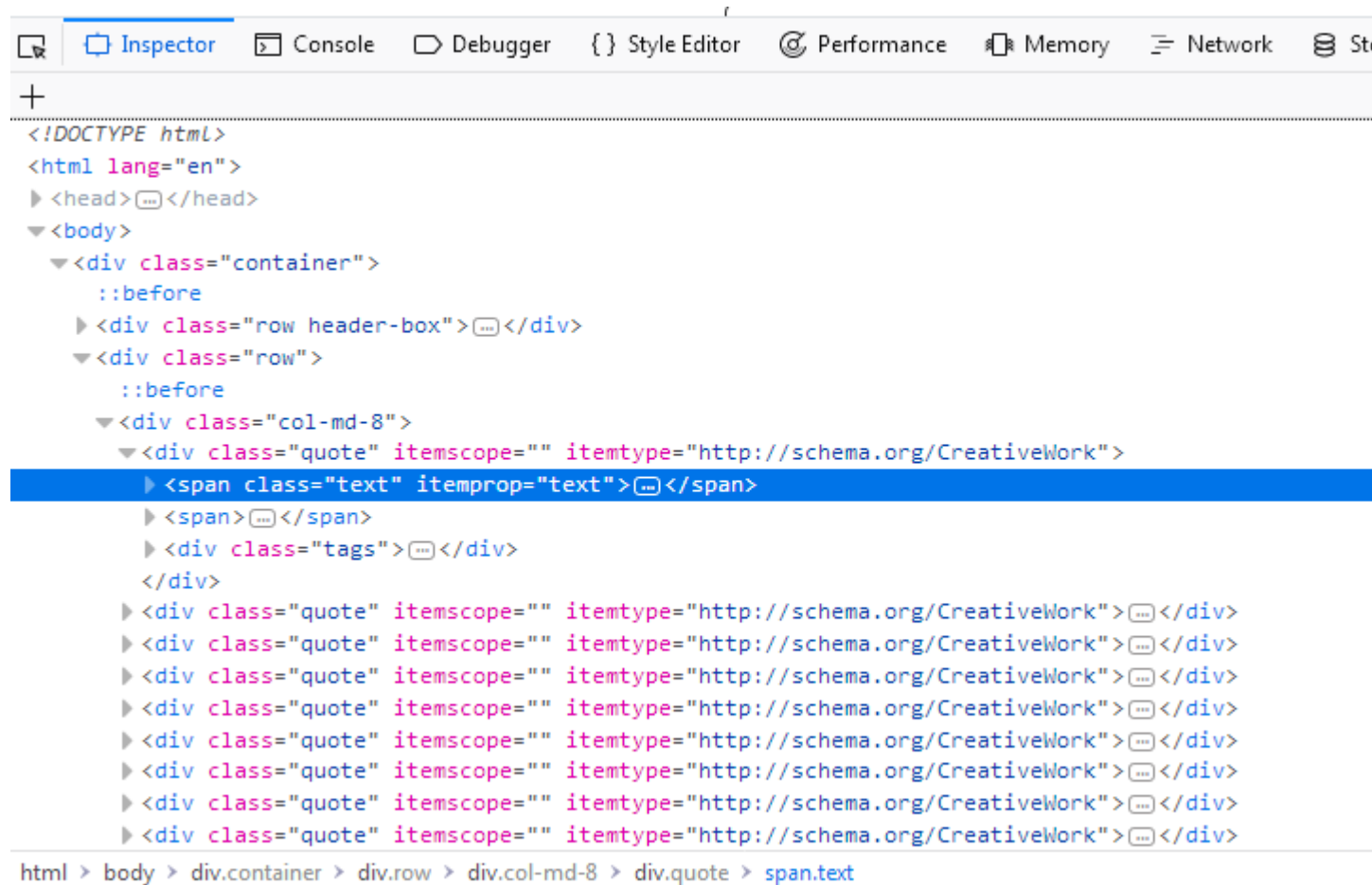
- Disable Javascript while inspecting the DOM looking for XPaths to be used in Scrapy (in the Developer Tools settings click *Disable JavaScript*)
- Never use full XPath paths, use relative and clever ones based on attributes (such as `id`, `class`, `width`, etc) or any identifying features like `contains(@href, 'image')`.
- Never include `<tbody>` elements in your XPath expressions unless you really know what you’re doing

### 5.6.2 Inspecting a website

By far the most handy feature of the Developer Tools is the *Inspector* feature, which allows you to inspect the underlying HTML code of any webpage. To demonstrate the Inspector, let’s look at the [quotes.toscrape.com](http://quotes.toscrape.com)-site.

On the site we have a total of ten quotes from various authors with specific tags, as well as the Top Ten Tags. Let's say we want to extract all the quotes on this page, without any meta-information about authors, tags, etc.

Instead of viewing the whole source code for the page, we can simply right click on a quote and select *Inspect Element* (Q), which opens up the *Inspector*. In it you should see something like this:



The interesting part for us is this:

```
<div class="quote" itemscope="" itemtype="http://schema.org/CreativeWork">
  <span class="text" itemprop="text">(...)</span>
  <span>(...)</span>
  <div class="tags">(...)</div>
</div>
```

If you hover over the first `div` directly above the `span` tag highlighted in the screenshot, you'll see that the corresponding section of the webpage gets highlighted as well. So now we have a section, but we can't find our quote text anywhere.

The advantage of the *Inspector* is that it automatically expands and collapses sections and tags of a webpage, which greatly improves readability. You can expand and collapse a tag by clicking on the arrow in front of it or by double clicking directly on the tag. If we expand the `span` tag with the `class="text"` we will see the quote-text we clicked on. The *Inspector* lets you copy XPath to selected elements. Let's try it out: Right-click on the `span` tag, select `Copy > XPath` and paste it in the scrapy shell like so:

```
$ scrapy shell "http://quotes.toscrape.com/"
```

(continues on next page)

(continued from previous page)

```
(...)
>>> response.xpath('/html/body/div/div[2]/div[1]/div[1]/span[1]/text()').getall()
['"The world as we have created it is a process of our thinking. It cannot be changed_
↳without changing our thinking."']
```

Adding `text()` at the end we are able to extract the first quote with this basic selector. But this XPath is not really that clever. All it does is go down a desired path in the source code starting from `html`. So let's see if we can refine our XPath a bit:

If we check the *Inspector* again we'll see that directly beneath our expanded `div` tag we have nine identical `div` tags, each with the same attributes as our first. If we expand any of them, we'll see the same structure as with our first quote: Two `span` tags and one `div` tag. We can expand each `span` tag with the `class="text"` inside our `div` tags and see each quote:

```
<div class="quote" itemscope="" itemtype="http://schema.org/CreativeWork">
  <span class="text" itemprop="text">
    "The world as we have created it is a process of our thinking. It cannot be_
↳changed without changing our thinking."
  </span>
  <span>(...)</span>
  <div class="tags">(...)</div>
</div>
```

With this knowledge we can refine our XPath: Instead of a path to follow, we'll simply select all `span` tags with the `class="text"` by using the `has-class-extension`:

```
>>> response.xpath('///span[has-class("text")]/text()').getall()
['"The world as we have created it is a process of our thinking. It cannot be changed_
↳without changing our thinking."',
 '"It is our choices, Harry, that show what we truly are, far more than our abilities.
↳"',
 '"There are only two ways to live your life. One is as though nothing is a miracle._
↳The other is as though everything is a miracle."',
 (...)]
```

And with one simple, cleverer XPath we are able to extract all quotes from the page. We could have constructed a loop over our first XPath to increase the number of the last `div`, but this would have been unnecessarily complex and by simply constructing an XPath with `has-class("text")` we were able to extract all quotes in one line.

The *Inspector* has a lot of other helpful features, such as searching in the source code or directly scrolling to an element you selected. Let's demonstrate a use case:

Say you want to find the `Next` button on the page. Type `Next` into the search bar on the top right of the *Inspector*. You should get two results. The first is a `li` tag with the `class="text"`, the second the text of an `a` tag. Right click on the `a` tag and select `Scroll into View`. If you hover over the tag, you'll see the button highlighted. From here we could easily create a *Link Extractor* to follow the pagination. On a simple site such as this, there may not be the need to find an element visually but the `Scroll into View` function can be quite useful on complex sites.

Note that the search bar can also be used to search for and test CSS selectors. For example, you could search for `span.text` to find all quote texts. Instead of a full text search, this searches for exactly the `span` tag with the `class="text"` in the page.

### 5.6.3 The Network-tool

While scraping you may come across dynamic webpages where some parts of the page are loaded dynamically through multiple requests. While this can be quite tricky, the *Network-tool* in the Developer Tools greatly facilitates this task.

To demonstrate the Network-tool, let's take a look at the page [quotes.toscrape.com/scroll](http://quotes.toscrape.com/scroll).

The page is quite similar to the basic [quotes.toscrape.com](http://quotes.toscrape.com)-page, but instead of the above-mentioned `Next` button, the page automatically loads new quotes when you scroll to the bottom. We could go ahead and try out different XPaths directly, but instead we'll check another quite useful command from the scrapy shell:

```
$ scrapy shell "quotes.toscrape.com/scroll"
(...)
>>> view(response)
```

A browser window should open with the webpage but with one crucial difference: Instead of the quotes we just see a greenish bar with the word `Loading...`

## Quotes to Scrape

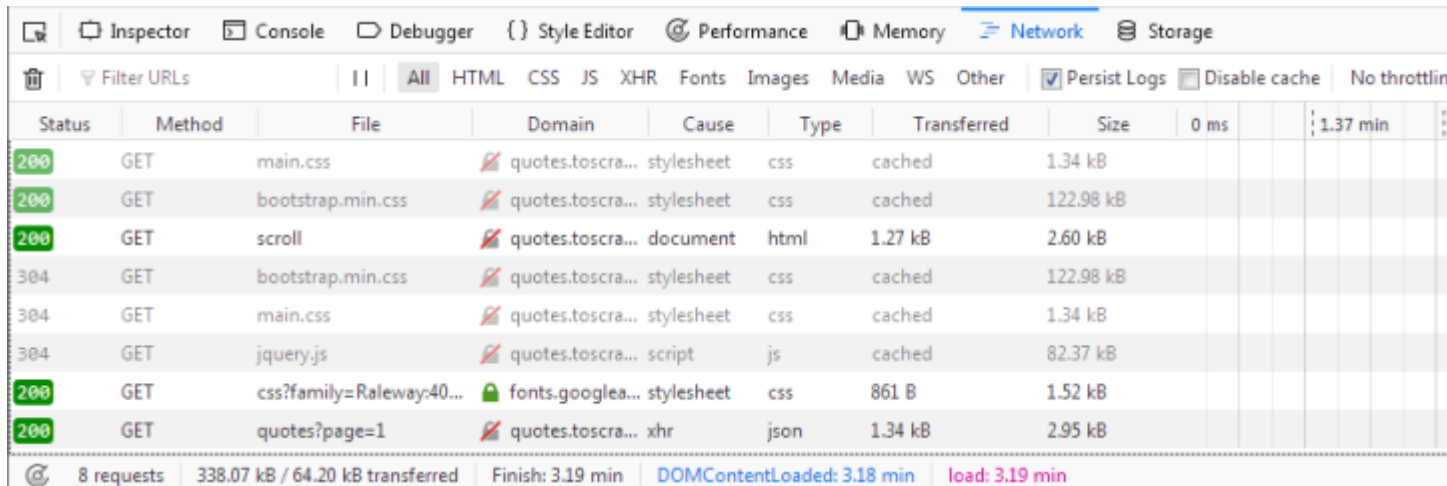
Log

Loading...

The `view(response)` command let's us view the response our shell or later our spider receives from the server. Here we see that some basic template is loaded which includes the title, the login-button and the footer, but the quotes are missing. This tells us that the quotes are being loaded from a different request than `quotes.toscrape.com/scroll`.

If you click on the `Network` tab, you will probably only see two entries. The first thing we do is enable persistent logs by clicking on `Persist Logs`. If this option is disabled, the log is automatically cleared each time you navigate to a different page. Enabling this option is a good default, since it gives us control on when to clear the logs.

If we reload the page now, you'll see the log get populated with six new requests.



Status	Method	File	Domain	Cause	Type	Transferred	Size	0 ms	1.37 min
200	GET	main.css	quotes.toscra...	stylesheet	css	cached	1.34 kB		
200	GET	bootstrap.min.css	quotes.toscra...	stylesheet	css	cached	122.98 kB		
200	GET	scroll	quotes.toscra...	document	html	1.27 kB	2.60 kB		
304	GET	bootstrap.min.css	quotes.toscra...	stylesheet	css	cached	122.98 kB		
304	GET	main.css	quotes.toscra...	stylesheet	css	cached	1.34 kB		
304	GET	jquery.js	quotes.toscra...	script	js	cached	82.37 kB		
200	GET	css?family=Raleway:40...	fonts.googlea...	stylesheet	css	861 B	1.52 kB		
200	GET	quotes?page=1	quotes.toscra...	xhr	json	1.34 kB	2.95 kB		

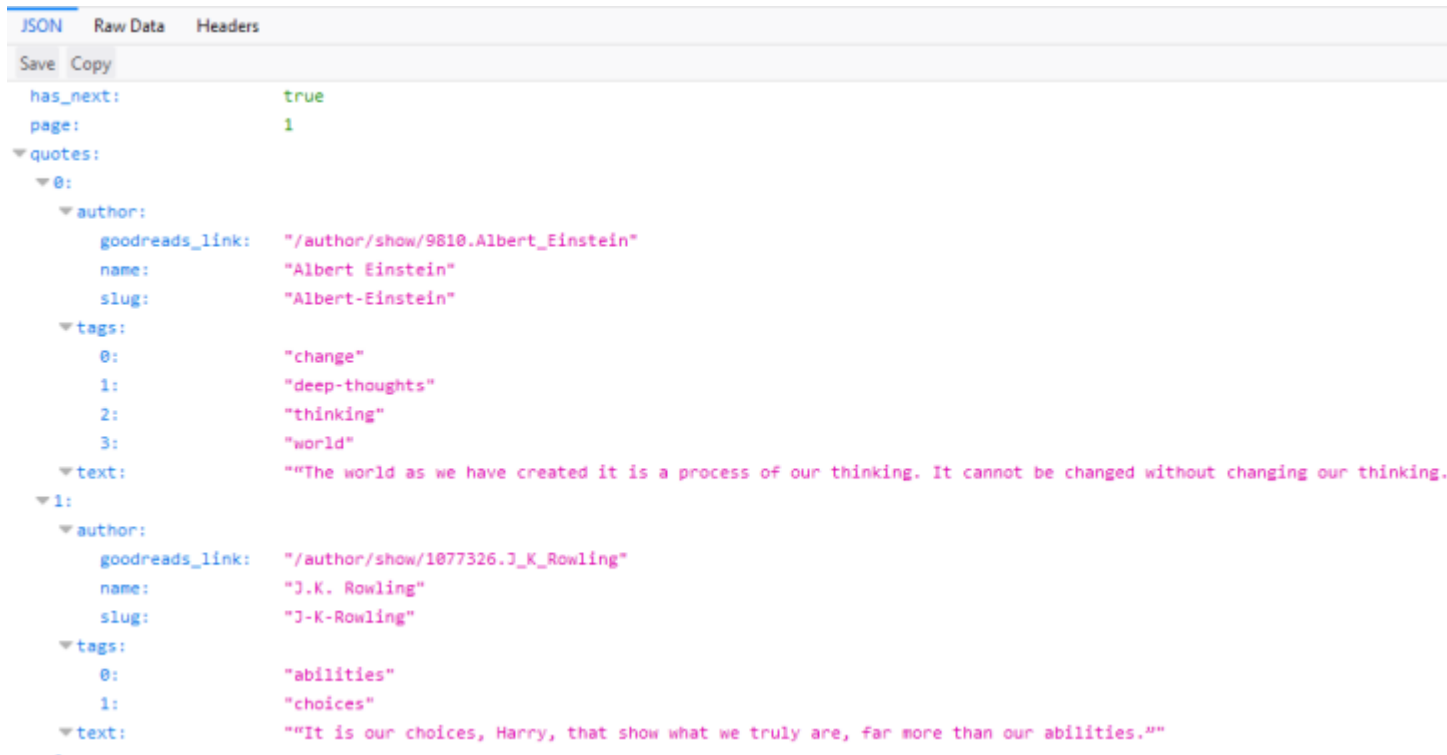
8 requests | 338.07 kB / 64.20 kB transferred | Finish: 3.19 min | DOMContentLoaded: 3.18 min | load: 3.19 min

Here we see every request that has been made when reloading the page and can inspect each request and its response. So let's find out where our quotes are coming from:

First click on the request with the name `scroll`. On the right you can now inspect the request. In `Headers` you'll find details about the request headers, such as the URL, the method, the IP-address, and so on. We'll ignore the other tabs and click directly on `Response`.

What you should see in the `Preview` pane is the rendered HTML-code, that is exactly what we saw when we called `view(response)` in the shell. Accordingly the type of the request in the log is `html`. The other requests have types like `css` or `js`, but what interests us is the one request called `quotes?page=1` with the type `json`.

If we click on this request, we see that the request URL is `http://quotes.toscrape.com/api/quotes?page=1` and the response is a JSON-object that contains our quotes. We can also right-click on the request and open `Open in new tab` to get a better overview.



JSON	Raw Data	Headers
Save Copy		
has_next:	true	
page:	1	
quotes:		
0:		
author:		
goodreads_link:	"/author/show/9810.Albert_Einstein"	
name:	"Albert Einstein"	
slug:	"Albert-Einstein"	
tags:		
0:	"change"	
1:	"deep-thoughts"	
2:	"thinking"	
3:	"world"	
text:	"The world as we have created it is a process of our thinking. It cannot be changed without changing our thinking."	
1:		
author:		
goodreads_link:	"/author/show/1077326.J_K_Rowling"	
name:	"J.K. Rowling"	
slug:	"J-K-Rowling"	
tags:		
0:	"abilities"	
1:	"choices"	
text:	"It is our choices, Harry, that show what we truly are, far more than our abilities."	

With this response we can now easily parse the JSON-object and also request each page to get every quote on the site:

```
import scrapy
import json

class QuoteSpider(scrapy.Spider):
    name = 'quote'
    allowed_domains = ['quotes.toscrape.com']
    page = 1
    start_urls = ['http://quotes.toscrape.com/api/quotes?page=1']

    def parse(self, response):
        data = json.loads(response.text)
        for quote in data["quotes"]:
            yield {"quote": quote["text"]}
        if data["has_next"]:
            self.page += 1
            url = "http://quotes.toscrape.com/api/quotes?page={}".format(self.page)
            yield scrapy.Request(url=url, callback=self.parse)
```

This spider starts at the first page of the quotes-API. With each response, we parse the `response.text` and assign it to `data`. This lets us operate on the JSON-object like on a Python dictionary. We iterate through the quotes and print out the `quote["text"]`. If the handy `has_next` element is `true` (try loading [quotes.toscrape.com/api/quotes?page=10](http://quotes.toscrape.com/api/quotes?page=10) in your browser or a page-number greater than 10), we increment the `page` attribute and `yield` a new request, inserting the incremented page-number into our `url`.

You can see that with a few inspections in the *Network*-tool we were able to easily replicate the dynamic requests of the scrolling functionality of the page. Crawling dynamic pages can be quite daunting and pages can be very complex, but it (mostly) boils down to identifying the correct request and replicating it in your spider.

## 5.7 Debugging memory leaks

In Scrapy, objects such as Requests, Responses and Items have a finite lifetime: they are created, used for a while, and finally destroyed.

From all those objects, the Request is probably the one with the longest lifetime, as it stays waiting in the Scheduler queue until it's time to process it. For more info see [Architecture overview](#).

As these Scrapy objects have a (rather long) lifetime, there is always the risk of accumulating them in memory without releasing them properly and thus causing what is known as a “memory leak”.

To help debugging memory leaks, Scrapy provides a built-in mechanism for tracking objects references called [trackref](#), and you can also use a third-party library called [Guppy](#) for more advanced memory debugging (see below for more info). Both mechanisms must be used from the *Telnet Console*.

### 5.7.1 Common causes of memory leaks

It happens quite often (sometimes by accident, sometimes on purpose) that the Scrapy developer passes objects referenced in Requests (for example, using the `meta` attribute or the request callback function) and that effectively bounds the lifetime of those referenced objects to the lifetime of the Request. This is, by far, the most common cause of memory leaks in Scrapy projects, and a quite difficult one to debug for newcomers.

In big projects, the spiders are typically written by different people and some of those spiders could be “leaking” and thus affecting the rest of the other (well-written) spiders when they get to run concurrently, which, in turn, affects the whole crawling process.

The leak could also come from a custom middleware, pipeline or extension that you have written, if you are not releasing the (previously allocated) resources properly. For example, allocating resources on `:signal:'spider_opened'` but not releasing them on `:signal:'spider_closed'` may cause problems if you're running *multiple spiders per process*.

## Too Many Requests?

By default Scrapy keeps the request queue in memory; it includes `Request` objects and all objects referenced in Request attributes (e.g. in `meta`). While not necessarily a leak, this can take a lot of memory. Enabling *persistent job queue* could help keeping memory usage in control.

### 5.7.2 Debugging memory leaks with `trackref`

`trackref` is a module provided by Scrapy to debug the most common cases of memory leaks. It basically tracks the references to all live Requests, Responses, Item and Selector objects.

You can enter the telnet console and inspect how many objects (of the classes mentioned above) are currently alive using the `prefs()` function which is an alias to the `print_live_refs()` function:

```
telnet localhost 6023

>>> prefs()
Live References

ExampleSpider          1   oldest: 15s ago
HtmlResponse          10  oldest: 1s ago
Selector               2   oldest: 0s ago
FormRequest           878  oldest: 7s ago
```

As you can see, that report also shows the “age” of the oldest object in each class. If you're running multiple spiders per process chances are you can figure out which spider is leaking by looking at the oldest request or response. You can get the oldest object of each class using the `get_oldest()` function (from the telnet console).

## Which objects are tracked?

The objects tracked by `trackrefs` are all from these classes (and all its subclasses):

- `scrapy.http.Request`
- `scrapy.http.Response`
- `scrapy.item.Item`
- `scrapy.selector.Selector`
- `scrapy.spiders.Spider`

## A real example

Let's see a concrete example of a hypothetical case of memory leaks. Suppose we have some spider with a line similar to this one:

```
return Request("http://www.somenastyspider.com/product.php?pid=%d" % product_id,
               callback=self.parse, meta={referer: response})
```



That line is passing a response reference inside a request which effectively ties the response lifetime to the requests' one, and that would definitely cause memory leaks.

Let's see how we can discover the cause (without knowing it a-priori, of course) by using the `trackref` tool.

After the crawler is running for a few minutes and we notice its memory usage has grown a lot, we can enter its telnet console and check the live references:

```
>>> prefs()
Live References

SomenastySpider          1   oldest: 15s ago
HtmlResponse             3890  oldest: 265s ago
Selector                  2   oldest: 0s ago
Request                   3878  oldest: 250s ago
```

The fact that there are so many live responses (and that they're so old) is definitely suspicious, as responses should have a relatively short lifetime compared to Requests. The number of responses is similar to the number of requests, so it looks like they are tied in a some way. We can now go and check the code of the spider to discover the nasty line that is generating the leaks (passing response references inside requests).

Sometimes extra information about live objects can be helpful. Let's check the oldest response:

```
>>> from scrapy.utils.trackref import get_oldest
>>> r = get_oldest('HtmlResponse')
>>> r.url
'http://www.somenastyspider.com/product.php?pid=123'
```

If you want to iterate over all objects, instead of getting the oldest one, you can use the `scrapy.utils.trackref.iter_all()` function:

```
>>> from scrapy.utils.trackref import iter_all
>>> [r.url for r in iter_all('HtmlResponse')]
['http://www.somenastyspider.com/product.php?pid=123',
 'http://www.somenastyspider.com/product.php?pid=584',
 ...]
```

## Too many spiders?

If your project has too many spiders executed in parallel, the output of `prefs()` can be difficult to read. For this reason, that function has a `ignore` argument which can be used to ignore a particular class (and all its subclasses). For example, this won't show any live references to spiders:

```
>>> from scrapy.spiders import Spider
>>> prefs(ignore=Spider)
```

## scrapy.utils.trackref module

Here are the functions available in the `trackref` module.

**class** scrapy.utils.trackref.object\_ref

Inherit from this class (instead of object) if you want to track live instances with the `trackref` module.

scrapy.utils.trackref.print\_live\_refs(*class\_name*, *ignore=NoneType*)

Print a report of live references, grouped by class name.

**Parameters** `ignore` (*class or classes tuple*) – if given, all objects from the specified class (or tuple of classes) will be ignored.

`scrapy.utils.trackref.get_oldest(class_name)`

Return the oldest object alive with the given class name, or `None` if none is found. Use `print_live_refs()` first to get a list of all tracked live objects per class name.

`scrapy.utils.trackref.iter_all(class_name)`

Return an iterator over all objects alive with the given class name, or `None` if none is found. Use `print_live_refs()` first to get a list of all tracked live objects per class name.

### 5.7.3 Debugging memory leaks with Guppy

`trackref` provides a very convenient mechanism for tracking down memory leaks, but it only keeps track of the objects that are more likely to cause memory leaks (Requests, Responses, Items, and Selectors). However, there are other cases where the memory leaks could come from other (more or less obscure) objects. If this is your case, and you can't find your leaks using `trackref`, you still have another resource: the [Guppy library](#). If you're using Python3, see [Debugging memory leaks with muppy](#).

If you use `pip`, you can install Guppy with the following command:

```
pip install guppy
```

The telnet console also comes with a built-in shortcut (`hpy`) for accessing Guppy heap objects. Here's an example to view all Python objects available in the heap using Guppy:

```
>>> x = hpy.heap()
>>> x.bytype
Partition of a set of 297033 objects. Total size = 52587824 bytes.
  Index  Count    %      Size  % Cumulative  % Type
    0    22307    8  16423880  31  16423880  31 dict
    1   122285   41  12441544  24  28865424  55 str
    2    68346   23   5966696  11  34832120  66 tuple
    3      227    0   5836528  11  40668648  77 unicode
    4    2461    1   2222272   4  42890920  82 type
    5   16870    6   2024400   4  44915320  85 function
    6   13949    5   1673880   3  46589200  89 types.CodeType
    7   13422    5   1653104   3  48242304  92 list
    8    3735    1   1173680   2  49415984  94 _sre.SRE_Pattern
    9    1209    0    456936   1  49872920  95 scrapy.http.headers.Headers
<1676 more rows. Type e.g. '_.more' to view.>
```

You can see that most space is used by dicts. Then, if you want to see from which attribute those dicts are referenced, you could do:

```
>>> x.bytype[0].byvia
Partition of a set of 22307 objects. Total size = 16423880 bytes.
  Index  Count    %      Size  % Cumulative  % Referred Via:
    0   10982   49   9416336  57   9416336  57 '.__dict__'
    1    1820    8   2681504  16  12097840  74 '.__dict__', '.func_globals'
    2    3097   14   1122904   7  13220744  80
    3     990    4    277200   2  13497944  82 "['cookies']"
    4     987    4    276360   2  13774304  84 "['cache']"
    5     985    4    275800   2  14050104  86 "['meta']"
    6     897    4    251160   2  14301264  87 '[2]'
    7        1    0    196888   1  14498152  88 "['moduleDict']", "['modules']"
    8     672    3    188160   1  14686312  89 "['cb_kwargs']"
```

(continues on next page)

(continued from previous page)

```

      9      27      0      155016      1      14841328      90      '[1]'
<333 more rows. Type e.g. '_.more' to view.>

```

As you can see, the Guppy module is very powerful but also requires some deep knowledge about Python internals. For more info about Guppy, refer to the [Guppy documentation](#).

## 5.7.4 Debugging memory leaks with muppy

If you're using Python 3, you can use muppy from [Pympler](#).

If you use `pip`, you can install muppy with the following command:

```
pip install Pympler
```

Here's an example to view all Python objects available in the heap using muppy:

```

>>> from pympler import muppy
>>> all_objects = muppy.get_objects()
>>> len(all_objects)
28667
>>> from pympler import summary
>>> sum1 = summary.summarize(all_objects)
>>> summary.print_(sum1)

```

	types	# objects	total size
	=====	=====	=====
	<class 'str	9822	1.10 MB
	<class 'dict	1658	856.62 KB
	<class 'type	436	443.60 KB
	<class 'code	2974	419.56 KB
	<class '_io.BufferedWriter	2	256.34 KB
	<class 'set	420	159.88 KB
	<class '_io.BufferedReader	1	128.17 KB
	<class 'wrapper_descriptor	1130	88.28 KB
	<class 'tuple	1304	86.57 KB
	<class 'weakref	1013	79.14 KB
	<class 'builtin_function_or_method	958	67.36 KB
	<class 'method_descriptor	865	60.82 KB
	<class 'abc.ABCMeta	62	59.96 KB
	<class 'list	446	58.52 KB
	<class 'int	1425	43.20 KB

For more info about muppy, refer to the [muppy documentation](#).

## 5.7.5 Leaks without leaks

Sometimes, you may notice that the memory usage of your Scrapy process will only increase, but never decrease. Unfortunately, this could happen even though neither Scrapy nor your project are leaking memory. This is due to a (not so well) known problem of Python, which may not return released memory to the operating system in some cases. For more information on this issue see:

- [Python Memory Management](#)
- [Python Memory Management Part 2](#)
- [Python Memory Management Part 3](#)

The improvements proposed by Evan Jones, which are detailed in [this paper](#), got merged in Python 2.5, but this only reduces the problem, it doesn't fix it completely. To quote the paper:

*Unfortunately, this patch can only free an arena if there are no more objects allocated in it anymore. This means that fragmentation is a large issue. An application could have many megabytes of free memory, scattered throughout all the arenas, but it will be unable to free any of it. This is a problem experienced by all memory allocators. The only way to solve it is to move to a compacting garbage collector, which is able to move objects in memory. This would require significant changes to the Python interpreter.*

To keep memory consumption reasonable you can split the job into several smaller jobs or enable *persistent job queue* and stop/start spider from time to time.

## 5.8 Downloading and processing files and images

Scrapy provides reusable *item pipelines* for downloading files attached to a particular item (for example, when you scrape products and also want to download their images locally). These pipelines share a bit of functionality and structure (we refer to them as media pipelines), but typically you'll either use the Files Pipeline or the Images Pipeline.

Both pipelines implement these features:

- Avoid re-downloading media that was downloaded recently
- Specifying where to store the media (filesystem directory, Amazon S3 bucket, Google Cloud Storage bucket)

The Images Pipeline has a few extra functions for processing images:

- Convert all downloaded images to a common format (JPG) and mode (RGB)
- Thumbnail generation
- Check images width/height to make sure they meet a minimum constraint

The pipelines also keep an internal queue of those media URLs which are currently being scheduled for download, and connect those responses that arrive containing the same media to that queue. This avoids downloading the same media more than once when it's shared by several items.

### 5.8.1 Using the Files Pipeline

The typical workflow, when using the `FilesPipeline` goes like this:

1. In a Spider, you scrape an item and put the URLs of the desired into a `file_urls` field.
2. The item is returned from the spider and goes to the item pipeline.
3. When the item reaches the `FilesPipeline`, the URLs in the `file_urls` field are scheduled for download using the standard Scrapy scheduler and downloader (which means the scheduler and downloader middlewares are reused), but with a higher priority, processing them before other pages are scraped. The item remains "locked" at that particular pipeline stage until the files have finish downloading (or fail for some reason).
4. When the files are downloaded, another field (`files`) will be populated with the results. This field will contain a list of dicts with information about the downloaded files, such as the downloaded path, the original scraped url (taken from the `file_urls` field), and the file checksum. The files in the list of the `files` field will retain the same order of the original `file_urls` field. If some file failed downloading, an error will be logged and the file won't be present in the `files` field.

## 5.8.2 Using the Images Pipeline

Using the *ImagesPipeline* is a lot like using the *FilesPipeline*, except the default field names used are different: you use `image_urls` for the image URLs of an item and it will populate an `images` field for the information about the downloaded images.

The advantage of using the *ImagesPipeline* for image files is that you can configure some extra functions like generating thumbnails and filtering the images based on their size.

The Images Pipeline uses *Pillow* for thumbnailing and normalizing images to JPEG/RGB format, so you need to install this library in order to use it. *Python Imaging Library* (PIL) should also work in most cases, but it is known to cause troubles in some setups, so we recommend to use *Pillow* instead of PIL.

## 5.8.3 Enabling your Media Pipeline

To enable your media pipeline you must first add it to your project **:setting:‘ITEM\_PIPELINES’** setting.

For Images Pipeline, use:

```
ITEM_PIPELINES = {'scrapy.pipelines.images.ImagesPipeline': 1}
```

For Files Pipeline, use:

```
ITEM_PIPELINES = {'scrapy.pipelines.files.FilesPipeline': 1}
```

---

**Note:** You can also use both the Files and Images Pipeline at the same time.

---

Then, configure the target storage setting to a valid value that will be used for storing the downloaded images. Otherwise the pipeline will remain disabled, even if you include it in the **:setting:‘ITEM\_PIPELINES’** setting.

For the Files Pipeline, set the **:setting:‘FILES\_STORE’** setting:

```
FILES_STORE = '/path/to/valid/dir'
```

For the Images Pipeline, set the **:setting:‘IMAGES\_STORE’** setting:

```
IMAGES_STORE = '/path/to/valid/dir'
```

## 5.8.4 Supported Storage

File system is currently the only officially supported storage, but there are also support for storing files in *Amazon S3* and *Google Cloud Storage*.

### File system storage

The files are stored using a *SHA1 hash* of their URLs for the file names.

For example, the following image URL:

```
http://www.example.com/image.jpg
```

Whose *SHA1 hash* is:

```
3afec3b4765f8f0a07b78f98c07b83f013567a0a
```

Will be downloaded and stored in the following file:

```
<IMAGES_STORE>/full/3afec3b4765f8f0a07b78f98c07b83f013567a0a.jpg
```

Where:

- `<IMAGES_STORE>` is the directory defined in **:setting:‘IMAGES\_STORE’** setting for the Images Pipeline.
- `full` is a sub-directory to separate full images from thumbnails (if used). For more info see *Thumbnail generation for images*.

## Amazon S3 storage

**:setting:‘FILES\_STORE’** and **:setting:‘IMAGES\_STORE’** can represent an Amazon S3 bucket. Scrapy will automatically upload the files to the bucket.

For example, this is a valid **:setting:‘IMAGES\_STORE’** value:

```
IMAGES_STORE = 's3://bucket/images'
```

You can modify the Access Control List (ACL) policy used for the stored files, which is defined by the **:setting:‘FILES\_STORE\_S3\_ACL’** and **:setting:‘IMAGES\_STORE\_S3\_ACL’** settings. By default, the ACL is set to private. To make the files publicly available use the `public-read` policy:

```
IMAGES_STORE_S3_ACL = 'public-read'
```

For more information, see [canned ACLs](#) in the Amazon S3 Developer Guide.

Because Scrapy uses `boto` / `botocore` internally you can also use other S3-like storages. Storages like self-hosted [Minio](#) or [s3.scality](#). All you need to do is set endpoint option in you Scrapy settings:

```
AWS_ENDPOINT_URL = 'http://minio.example.com:9000'
```

For self-hosting you also might feel the need not to use SSL and not to verify SSL connection:

```
AWS_USE_SSL = False # or True (None by default)
AWS_VERIFY = False # or True (None by default)
```

## Google Cloud Storage

**:setting:‘FILES\_STORE’** and **:setting:‘IMAGES\_STORE’** can represent a Google Cloud Storage bucket. Scrapy will automatically upload the files to the bucket. (requires [google-cloud-storage](#) )

For example, these are valid **:setting:‘IMAGES\_STORE’** and **:setting:‘GCS\_PROJECT\_ID’** settings:

```
IMAGES_STORE = 'gs://bucket/images/'
GCS_PROJECT_ID = 'project_id'
```

For information about authentication, see this [documentation](#).

You can modify the Access Control List (ACL) policy used for the stored files, which is defined by the **:setting:‘FILES\_STORE\_GCS\_ACL’** and **:setting:‘IMAGES\_STORE\_GCS\_ACL’** settings. By default, the ACL is set to `''` (empty string) which means that Cloud Storage applies the bucket’s default object ACL to the object. To make the files publicly available use the `publicRead` policy:

```
IMAGES_STORE_GCS_ACL = 'publicRead'
```

For more information, see [Predefined ACLs](#) in the Google Cloud Platform Developer Guide.

### 5.8.5 Usage example

In order to use a media pipeline first, *enable it*.

Then, if a spider returns a dict with the URLs key (`file_urls` or `image_urls`, for the Files or Images Pipeline respectively), the pipeline will put the results under respective key (`files` or `images`).

If you prefer to use *Item*, then define a custom item with the necessary fields, like in this example for Images Pipeline:

```
import scrapy

class MyItem(scrapy.Item):

    # ... other item fields ...
    image_urls = scrapy.Field()
    images = scrapy.Field()
```

If you want to use another field name for the URLs key or for the results key, it is also possible to override it.

For the Files Pipeline, set **:setting:'FILES\_URLS\_FIELD'** and/or **:setting:'FILES\_RESULT\_FIELD'** settings:

```
FILES_URLS_FIELD = 'field_name_for_your_files_urls'
FILES_RESULT_FIELD = 'field_name_for_your_processed_files'
```

For the Images Pipeline, set **:setting:'IMAGES\_URLS\_FIELD'** and/or **:setting:'IMAGES\_RESULT\_FIELD'** settings:

```
IMAGES_URLS_FIELD = 'field_name_for_your_images_urls'
IMAGES_RESULT_FIELD = 'field_name_for_your_processed_images'
```

If you need something more complex and want to override the custom pipeline behaviour, see *Extending the Media Pipelines*.

If you have multiple image pipelines inheriting from ImagePipeline and you want to have different settings in different pipelines you can set setting keys preceded with uppercase name of your pipeline class. E.g. if your pipeline is called MyPipeline and you want to have custom IMAGES\_URLS\_FIELD you define setting MYP-PIPELINE\_IMAGES\_URLS\_FIELD and your custom settings will be used.

### 5.8.6 Additional features

#### File expiration

The Image Pipeline avoids downloading files that were downloaded recently. To adjust this retention delay use the **:setting:'FILES\_EXPIRES'** setting (or **:setting:'IMAGES\_EXPIRES'**, in case of Images Pipeline), which specifies the delay in number of days:

```
# 120 days of delay for files expiration
FILES_EXPIRES = 120

# 30 days of delay for images expiration
IMAGES_EXPIRES = 30
```

The default value for both settings is 90 days.

If you have pipeline that subclasses FilesPipeline and you'd like to have different setting for it you can set setting keys preceded by uppercase class name. E.g. given pipeline class called MyPipeline you can set setting key:

```
MYPIPELINE_FILES_EXPIRES = 180
```

and pipeline class MyPipeline will have expiration time set to 180.

## Thumbnail generation for images

The Images Pipeline can automatically create thumbnails of the downloaded images.

In order use this feature, you must set **:setting:'IMAGES\_THUMBS'** to a dictionary where the keys are the thumbnail names and the values are their dimensions.

For example:

```
IMAGES_THUMBS = {
    'small': (50, 50),
    'big': (270, 270),
}
```

When you use this feature, the Images Pipeline will create thumbnails of the each specified size with this format:

```
<IMAGES_STORE>/thumbs/<size_name>/<image_id>.jpg
```

Where:

- `<size_name>` is the one specified in the **:setting:'IMAGES\_THUMBS'** dictionary keys (small, big, etc)
- `<image_id>` is the [SHA1 hash](#) of the image url

Example of image files stored using small and big thumbnail names:

```
<IMAGES_STORE>/full/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
<IMAGES_STORE>/thumbs/small/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
<IMAGES_STORE>/thumbs/big/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
```

The first one is the full image, as downloaded from the site.

## Filtering out small images

When using the Images Pipeline, you can drop images which are too small, by specifying the minimum allowed size in the **:setting:'IMAGES\_MIN\_HEIGHT'** and **:setting:'IMAGES\_MIN\_WIDTH'** settings.

For example:

```
IMAGES_MIN_HEIGHT = 110
IMAGES_MIN_WIDTH = 110
```

---

**Note:** The size constraints don't affect thumbnail generation at all.

---

It is possible to set just one size constraint or both. When setting both of them, only images that satisfy both minimum sizes will be saved. For the above example, images of sizes (105 x 105) or (105 x 200) or (200 x 105) will all be dropped because at least one dimension is shorter than the constraint.

By default, there are no size constraints, so all images are processed.



## Allowing redirections

By default media pipelines ignore redirects, i.e. an HTTP redirection to a media file URL request will mean the media download is considered failed.

To handle media redirections, set this setting to `True`:

```
MEDIA_ALLOW_REDIRECTS = True
```

## 5.8.7 Extending the Media Pipelines

See here the methods that you can override in your custom Files Pipeline:

```
class scrapy.pipelines.files.FilesPipeline
```

**get\_media\_requests** (*item*, *info*)

As seen on the workflow, the pipeline will get the URLs of the images to download from the item. In order to do this, you can override the `get_media_requests()` method and return a Request for each file URL:

```
def get_media_requests(self, item, info):
    for file_url in item['file_urls']:
        yield scrapy.Request(file_url)
```

Those requests will be processed by the pipeline and, when they have finished downloading, the results will be sent to the `item_completed()` method, as a list of 2-element tuples. Each tuple will contain (success, file\_info\_or\_error) where:

- success is a boolean which is `True` if the image was downloaded successfully or `False` if it failed for some reason
- file\_info\_or\_error is a dict containing the following keys (if success is `True`) or a `Twisted Failure` if there was a problem.
  - url - the url where the file was downloaded from. This is the url of the request returned from the `get_media_requests()` method.
  - path - the path (relative to `:setting:'FILES_STORE'`) where the file was stored
  - checksum - a `MD5` hash of the image contents

The list of tuples received by `item_completed()` is guaranteed to retain the same order of the requests returned from the `get_media_requests()` method.

Here's a typical value of the results argument:

```
[ (True,
  {'checksum': '2b00042f7481c7b056c4b410d28f33cf',
   'path': 'full/0a79c461a4062ac383dc4fade7bc09f1384a3910.jpg',
   'url': 'http://www.example.com/files/product1.pdf'}),
  (False,
   Failure(...)) ]
```

By default the `get_media_requests()` method returns `None` which means there are no files to download for the item.

**item\_completed** (*results*, *item*, *info*)

The `FilesPipeline.item_completed()` method called when all file requests for a single item have completed (either finished downloading, or failed for some reason).

The `item_completed()` method must return the output that will be sent to subsequent item pipeline stages, so you must return (or drop) the item, as you would in any pipeline.

Here is an example of the `item_completed()` method where we store the downloaded file paths (passed in results) in the `file_paths` item field, and we drop the item if it doesn't contain any files:

```
from scrapy.exceptions import DropItem

def item_completed(self, results, item, info):
    file_paths = [x['path'] for ok, x in results if ok]
    if not file_paths:
        raise DropItem("Item contains no files")
    item['file_paths'] = file_paths
    return item
```

By default, the `item_completed()` method returns the item.

See here the methods that you can override in your custom Images Pipeline:

**class** scrapy.pipelines.images.ImagesPipeline

The *ImagesPipeline* is an extension of the *FilesPipeline*, customizing the field names and adding custom behavior for images.

**get\_media\_requests** (item, info)

Works the same way as *FilesPipeline.get\_media\_requests()* method, but using a different field name for image urls.

Must return a Request for each image URL.

**item\_completed** (results, item, info)

The *ImagesPipeline.item\_completed()* method is called when all image requests for a single item have completed (either finished downloading, or failed for some reason).

Works the same way as *FilesPipeline.item\_completed()* method, but using a different field names for storing image downloading results.

By default, the `item_completed()` method returns the item.

### 5.8.8 Custom Images pipeline example

Here is a full example of the Images Pipeline whose methods are exemplified above:

```
import scrapy
from scrapy.pipelines.images import ImagesPipeline
from scrapy.exceptions import DropItem

class MyImagesPipeline(ImagesPipeline):

    def get_media_requests(self, item, info):
        for image_url in item['image_urls']:
            yield scrapy.Request(image_url)

    def item_completed(self, results, item, info):
        image_paths = [x['path'] for ok, x in results if ok]
        if not image_paths:
            raise DropItem("Item contains no images")
        item['image_paths'] = image_paths
        return item
```

## 5.9 Deploying Spiders

This section describes the different options you have for deploying your Scrapy spiders to run them on a regular basis. Running Scrapy spiders in your local machine is very convenient for the (early) development stage, but not so much when you need to execute long-running spiders or move spiders to run in production continuously. This is where the solutions for deploying Scrapy spiders come in.

Popular choices for deploying Scrapy spiders are:

- *Scrapyd* (open source)
- *Scrapy Cloud* (cloud-based)

### 5.9.1 Deploying to a Scrapyd Server

*Scrapyd* is an open source application to run Scrapy spiders. It provides a server with HTTP API, capable of running and monitoring Scrapy spiders.

To deploy spiders to Scrapyd, you can use the `scrapyd-deploy` tool provided by the `scrapyd-client` package. Please refer to the [scrapyd-deploy documentation](#) for more information.

Scrapyd is maintained by some of the Scrapy developers.

### 5.9.2 Deploying to Scrapy Cloud

*Scrapy Cloud* is a hosted, cloud-based service by *Scrapinghub*, the company behind Scrapy.

Scrapy Cloud removes the need to setup and monitor servers and provides a nice UI to manage spiders and review scraped items, logs and stats.

To deploy spiders to Scrapy Cloud you can use the `shub` command line tool. Please refer to the [Scrapy Cloud documentation](#) for more information.

Scrapy Cloud is compatible with Scrapyd and one can switch between them as needed - the configuration is read from the `scrapy.cfg` file just like `scrapyd-deploy`.

## 5.10 AutoThrottle extension

This is an extension for automatically throttling crawling speed based on load of both the Scrapy server and the website you are crawling.

### 5.10.1 Design goals

1. be nicer to sites instead of using default download delay of zero
2. automatically adjust scrapy to the optimum crawling speed, so the user doesn't have to tune the download delays to find the optimum one. The user only needs to specify the maximum concurrent requests it allows, and the extension does the rest.

### 5.10.2 How it works

AutoThrottle extension adjusts download delays dynamically to make spider send **:setting:‘AUTOTHROTTLE\_TARGET\_CONCURRENCY‘** concurrent requests on average to each remote website.

It uses download latency to compute the delays. The main idea is the following: if a server needs `latency` seconds to respond, a client should send a request each `latency/N` seconds to have `N` requests processed in parallel.

Instead of adjusting the delays one can just set a small fixed download delay and impose hard limits on concurrency using **:setting:‘CONCURRENT\_REQUESTS\_PER\_DOMAIN‘** or **:setting:‘CONCURRENT\_REQUESTS\_PER\_IP‘** options. It will provide a similar effect, but there are some important differences:

- because the download delay is small there will be occasional bursts of requests;
- often non-200 (error) responses can be returned faster than regular responses, so with a small download delay and a hard concurrency limit crawler will be sending requests to server faster when server starts to return errors. But this is an opposite of what crawler should do - in case of errors it makes more sense to slow down: these errors may be caused by the high request rate.

AutoThrottle doesn't have these issues.

### 5.10.3 Throttling algorithm

AutoThrottle algorithm adjusts download delays based on the following rules:

1. spiders always start with a download delay of **:setting:‘AUTOTHROTTLE\_START\_DELAY‘**;
2. when a response is received, the target download delay is calculated as `latency / N` where `latency` is a latency of the response, and `N` is **:setting:‘AUTOTHROTTLE\_TARGET\_CONCURRENCY‘**;
3. download delay for next requests is set to the average of previous download delay and the target download delay;
4. latencies of non-200 responses are not allowed to decrease the delay;
5. download delay can't become less than **:setting:‘DOWNLOAD\_DELAY‘** or greater than **:setting:‘AUTOTHROTTLE\_MAX\_DELAY‘**

---

**Note:** The AutoThrottle extension honours the standard Scrapy settings for concurrency and delay. This means that it will respect **:setting:‘CONCURRENT\_REQUESTS\_PER\_DOMAIN‘** and **:setting:‘CONCURRENT\_REQUESTS\_PER\_IP‘** options and never set a download delay lower than **:setting:‘DOWNLOAD\_DELAY‘**.

---

In Scrapy, the download latency is measured as the time elapsed between establishing the TCP connection and receiving the HTTP headers.

Note that these latencies are very hard to measure accurately in a cooperative multitasking environment because Scrapy may be busy processing a spider callback, for example, and unable to attend downloads. However, these latencies should still give a reasonable estimate of how busy Scrapy (and ultimately, the server) is, and this extension builds on that premise.

### 5.10.4 Settings

The settings used to control the AutoThrottle extension are:

- **:setting:‘AUTOTHROTTLE\_ENABLED‘**
- **:setting:‘AUTOTHROTTLE\_START\_DELAY‘**

- **:setting:‘AUTOTHROTTLER\_MAX\_DELAY‘**
- **:setting:‘AUTOTHROTTLER\_TARGET\_CONCURRENCY‘**
- **:setting:‘AUTOTHROTTLER\_DEBUG‘**
- **:setting:‘CONCURRENT\_REQUESTS\_PER\_DOMAIN‘**
- **:setting:‘CONCURRENT\_REQUESTS\_PER\_IP‘**
- **:setting:‘DOWNLOAD\_DELAY‘**

For more information see *How it works*.

## AUTOTHROTTLER\_ENABLED

Default: `False`

Enables the AutoThrottle extension.

## AUTOTHROTTLER\_START\_DELAY

Default: `5.0`

The initial download delay (in seconds).

## AUTOTHROTTLER\_MAX\_DELAY

Default: `60.0`

The maximum download delay (in seconds) to be set in case of high latencies.

## AUTOTHROTTLER\_TARGET\_CONCURRENCY

New in version 1.1.

Default: `1.0`

Average number of requests Scrapy should be sending in parallel to remote websites.

By default, AutoThrottle adjusts the delay to send a single concurrent request to each of the remote websites. Set this option to a higher value (e.g. `2.0`) to increase the throughput and the load on remote servers. A lower `AUTOTHROTTLER_TARGET_CONCURRENCY` value (e.g. `0.5`) makes the crawler more conservative and polite.

Note that **:setting:‘CONCURRENT\_REQUESTS\_PER\_DOMAIN‘** and **:setting:‘CONCURRENT\_REQUESTS\_PER\_IP‘** options are still respected when AutoThrottle extension is enabled. This means that if `AUTOTHROTTLER_TARGET_CONCURRENCY` is set to a value higher than **:setting:‘CONCURRENT\_REQUESTS\_PER\_DOMAIN‘** or **:setting:‘CONCURRENT\_REQUESTS\_PER\_IP‘**, the crawler won’t reach this number of concurrent requests.

At every given time point Scrapy can be sending more or less concurrent requests than `AUTOTHROTTLER_TARGET_CONCURRENCY`; it is a suggested value the crawler tries to approach, not a hard limit.

## AUTOTHROTTLE\_DEBUG

Default: False

Enable AutoThrottle debug mode which will display stats on every response received, so you can see how the throttling parameters are being adjusted in real time.

## 5.11 Benchmarking

New in version 0.17.

Scrapy comes with a simple benchmarking suite that spawns a local HTTP server and crawls it at the maximum possible speed. The goal of this benchmarking is to get an idea of how Scrapy performs in your hardware, in order to have a common baseline for comparisons. It uses a simple spider that does nothing and just follows links.

To run it use:

```
scrapy bench
```

You should see an output like this:

```
2016-12-16 21:18:48 [scrapy.utils.log] INFO: Scrapy 1.2.2 started (bot: quotesbot)
2016-12-16 21:18:48 [scrapy.utils.log] INFO: Overridden settings: {'CLOSESPIDER_
↳ TIMEOUT': 10, 'ROBOTSTXT_OBEY': True, 'SPIDER_MODULES': ['quotesbot.spiders'],
↳ 'LOGSTATS_INTERVAL': 1, 'BOT_NAME': 'quotesbot', 'LOG_LEVEL': 'INFO', 'NEWSPIDER_
↳ MODULE': 'quotesbot.spiders'}
2016-12-16 21:18:49 [scrapy.middleware] INFO: Enabled extensions:
['scrapy.extensions.closespider.CloseSpider',
 'scrapy.extensions.logstats.LogStats',
 'scrapy.extensions.telnet.TelnetConsole',
 'scrapy.extensions.corestats.CoreStats']
2016-12-16 21:18:49 [scrapy.middleware] INFO: Enabled downloader middlewares:
['scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware',
 'scrapy.downloadermiddlewares.httpauth.HttpAuthMiddleware',
 'scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware',
 'scrapy.downloadermiddlewares.defaultheaders.DefaultHeadersMiddleware',
 'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware',
 'scrapy.downloadermiddlewares.retry.RetryMiddleware',
 'scrapy.downloadermiddlewares.redirect.MetaRefreshMiddleware',
 'scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware',
 'scrapy.downloadermiddlewares.redirect.RedirectMiddleware',
 'scrapy.downloadermiddlewares.cookies.CookiesMiddleware',
 'scrapy.downloadermiddlewares.stats.DownloaderStats']
2016-12-16 21:18:49 [scrapy.middleware] INFO: Enabled spider middlewares:
['scrapy.spidermiddlewares.httperror.HttpErrorMiddleware',
 'scrapy.spidermiddlewares.offsite.OffsiteMiddleware',
 'scrapy.spidermiddlewares.referer.RefererMiddleware',
 'scrapy.spidermiddlewares.urllength.UrlLengthMiddleware',
 'scrapy.spidermiddlewares.depth.DepthMiddleware']
2016-12-16 21:18:49 [scrapy.middleware] INFO: Enabled item pipelines:
[]
2016-12-16 21:18:49 [scrapy.core.engine] INFO: Spider opened
2016-12-16 21:18:49 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pages/
↳ min), scraped 0 items (at 0 items/min)
2016-12-16 21:18:50 [scrapy.extensions.logstats] INFO: Crawled 70 pages (at 4200_
↳ pages/min), scraped 0 items (at 0 items/min)
```

(continues on next page)

(continued from previous page)

```

2016-12-16 21:18:51 [scrapy.extensions.logstats] INFO: Crawled 134 pages (at 3840_
↳pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:18:52 [scrapy.extensions.logstats] INFO: Crawled 198 pages (at 3840_
↳pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:18:53 [scrapy.extensions.logstats] INFO: Crawled 254 pages (at 3360_
↳pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:18:54 [scrapy.extensions.logstats] INFO: Crawled 302 pages (at 2880_
↳pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:18:55 [scrapy.extensions.logstats] INFO: Crawled 358 pages (at 3360_
↳pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:18:56 [scrapy.extensions.logstats] INFO: Crawled 406 pages (at 2880_
↳pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:18:57 [scrapy.extensions.logstats] INFO: Crawled 438 pages (at 1920_
↳pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:18:58 [scrapy.extensions.logstats] INFO: Crawled 470 pages (at 1920_
↳pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:18:59 [scrapy.core.engine] INFO: Closing spider (closespider_timeout)
2016-12-16 21:18:59 [scrapy.extensions.logstats] INFO: Crawled 518 pages (at 2880_
↳pages/min), scraped 0 items (at 0 items/min)
2016-12-16 21:19:00 [scrapy.statscollectors] INFO: Dumping Scrapy stats:
{'downloader/request_bytes': 229995,
 'downloader/request_count': 534,
 'downloader/request_method_count/GET': 534,
 'downloader/response_bytes': 1565504,
 'downloader/response_count': 534,
 'downloader/response_status_count/200': 534,
 'finish_reason': 'closespider_timeout',
 'finish_time': datetime.datetime(2016, 12, 16, 16, 19, 0, 647725),
 'log_count/INFO': 17,
 'request_depth_max': 19,
 'response_received_count': 534,
 'scheduler/dequeued': 533,
 'scheduler/dequeued/memory': 533,
 'scheduler/enqueued': 10661,
 'scheduler/enqueued/memory': 10661,
 'start_time': datetime.datetime(2016, 12, 16, 16, 18, 49, 799869)}
2016-12-16 21:19:00 [scrapy.core.engine] INFO: Spider closed (closespider_timeout)

```

That tells you that Scrapy is able to crawl about 3000 pages per minute in the hardware where you run it. Note that this is a very simple spider intended to follow links, any custom spider you write will probably do more stuff which results in slower crawl rates. How slower depends on how much your spider does and how well it's written.

In the future, more cases will be added to the benchmarking suite to cover other common scenarios.

## 5.12 Jobs: pausing and resuming crawls

Sometimes, for big sites, it's desirable to pause crawls and be able to resume them later.

Scrapy supports this functionality out of the box by providing the following facilities:

- a scheduler that persists scheduled requests on disk
- a duplicates filter that persists visited requests on disk
- an extension that keeps some spider state (key/value pairs) persistent between batches

### 5.12.1 Job directory

To enable persistence support you just need to define a *job directory* through the `JOBDIR` setting. This directory will be for storing all required data to keep the state of a single job (ie. a spider run). It's important to note that this directory must not be shared by different spiders, or even different jobs/runs of the same spider, as it's meant to be used for storing the state of a *single* job.

### 5.12.2 How to use it

To start a spider with persistence supported enabled, run it like this:

```
scrapy crawl somespider -s JOBDIR=crawls/somespider-1
```

Then, you can stop the spider safely at any time (by pressing Ctrl-C or sending a signal), and resume it later by issuing the same command:

```
scrapy crawl somespider -s JOBDIR=crawls/somespider-1
```

### 5.12.3 Keeping persistent state between batches

Sometimes you'll want to keep some persistent spider state between pause/resume batches. You can use the `spider.state` attribute for that, which should be a dict. There's a built-in extension that takes care of serializing, storing and loading that attribute from the job directory, when the spider starts and stops.

Here's an example of a callback that uses the spider state (other spider code is omitted for brevity):

```
def parse_item(self, response):
    # parse item here
    self.state['items_count'] = self.state.get('items_count', 0) + 1
```

### 5.12.4 Persistence gotchas

There are a few things to keep in mind if you want to be able to use the Scrapy persistence support:

#### Cookies expiration

Cookies may expire. So, if you don't resume your spider quickly the requests scheduled may no longer work. This won't be an issue if you spider doesn't rely on cookies.

#### Request serialization

Requests must be serializable by the *pickle* module, in order for persistence to work, so you should make sure that your requests are serializable.

The most common issue here is to use `lambda` functions on request callbacks that can't be persisted.

So, for example, this won't work:



```
def some_callback(self, response):
    somearg = 'test'
    return scrapy.Request('http://www.example.com', callback=lambda r: self.other_
↳callback(r, somearg))

def other_callback(self, response, somearg):
    print "the argument passed is:", somearg
```

But this will:

```
def some_callback(self, response):
    somearg = 'test'
    return scrapy.Request('http://www.example.com', callback=self.other_callback,
↳meta={'somearg': somearg})

def other_callback(self, response):
    somearg = response.meta['somearg']
    print "the argument passed is:", somearg
```

If you wish to log the requests that couldn't be serialized, you can set the **:setting:'SCHEDULER\_DEBUG'** setting to True in the project's settings page. It is False by default.

**Frequently Asked Questions** Get answers to most frequently asked questions.

**Debugging Spiders** Learn how to debug common problems of your scrapy spider.

**Spiders Contracts** Learn how to use contracts for testing your spiders.

**Common Practices** Get familiar with some Scrapy common practices.

**Broad Crawls** Tune Scrapy for crawling a lot domains in parallel.

**Using your browser's Developer Tools for scraping** Learn how to scrape with your browser's developer tools.

**Debugging memory leaks** Learn how to find and get rid of memory leaks in your crawler.

**Downloading and processing files and images** Download files and/or images associated with your scraped items.

**Deploying Spiders** Deploying your Scrapy spiders and run them in a remote server.

**AutoThrottle extension** Adjust crawl rate dynamically based on load.

**Benchmarking** Check how Scrapy performs on your hardware.

**Jobs: pausing and resuming crawls** Learn how to pause and resume crawls for large spiders.



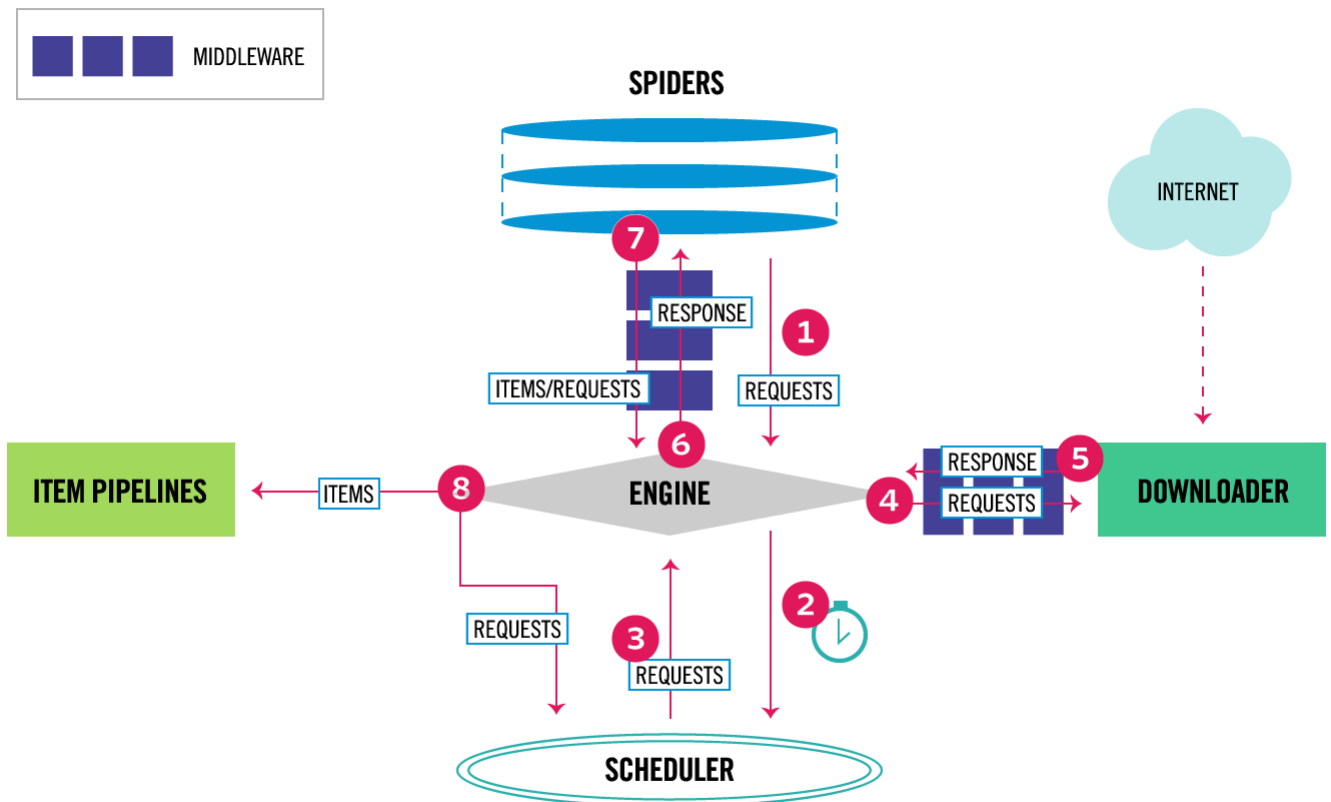
### 6.1 Architecture overview

This document describes the architecture of Scrapy and how its components interact.

#### 6.1.1 Overview

The following diagram shows an overview of the Scrapy architecture with its components and an outline of the data flow that takes place inside the system (shown by the red arrows). A brief description of the components is included below with links for more detailed information about them. The data flow is also described below.

### 6.1.2 Data flow



The data flow in Scrapy is controlled by the execution engine, and goes like this:

1. The *Engine* gets the initial Requests to crawl from the *Spider*.
2. The *Engine* schedules the Requests in the *Scheduler* and asks for the next Requests to crawl.
3. The *Scheduler* returns the next Requests to the *Engine*.
4. The *Engine* sends the Requests to the *Downloader*, passing through the *Downloader Middlewares* (see `process_request()`).
5. Once the page finishes downloading the *Downloader* generates a Response (with that page) and sends it to the Engine, passing through the *Downloader Middlewares* (see `process_response()`).
6. The *Engine* receives the Response from the *Downloader* and sends it to the *Spider* for processing, passing through the *Spider Middleware* (see `process_spider_input()`).
7. The *Spider* processes the Response and returns scraped items and new Requests (to follow) to the *Engine*, passing through the *Spider Middleware* (see `process_spider_output()`).
8. The *Engine* sends processed items to *Item Pipelines*, then send processed Requests to the *Scheduler* and asks for possible next Requests to crawl.
9. The process repeats (from step 1) until there are no more requests from the *Scheduler*.

### 6.1.3 Components

#### Scrapy Engine

The engine is responsible for controlling the data flow between all components of the system, and triggering events when certain actions occur. See the [Data Flow](#) section above for more details.

#### Scheduler

The Scheduler receives requests from the engine and enqueues them for feeding them later (also to the engine) when the engine requests them.

#### Downloader

The Downloader is responsible for fetching web pages and feeding them to the engine which, in turn, feeds them to the spiders.

#### Spiders

Spiders are custom classes written by Scrapy users to parse responses and extract items (aka scraped items) from them or additional requests to follow. For more information see [Spiders](#).

#### Item Pipeline

The Item Pipeline is responsible for processing the items once they have been extracted (or scraped) by the spiders. Typical tasks include cleansing, validation and persistence (like storing the item in a database). For more information see [Item Pipeline](#).

#### Downloader middlewares

Downloader middlewares are specific hooks that sit between the Engine and the Downloader and process requests when they pass from the Engine to the Downloader, and responses that pass from Downloader to the Engine.

Use a Downloader middleware if you need to do one of the following:

- process a request just before it is sent to the Downloader (i.e. right before Scrapy sends the request to the website);
- change received response before passing it to a spider;
- send a new Request instead of passing received response to a spider;
- pass response to a spider without fetching a web page;
- silently drop some requests.

For more information see [Downloader Middleware](#).

## Spider middlewares

Spider middlewares are specific hooks that sit between the Engine and the Spiders and are able to process spider input (responses) and output (items and requests).

Use a Spider middleware if you need to

- post-process output of spider callbacks - change/add/remove requests or items;
- post-process start\_requests;
- handle spider exceptions;
- call errback instead of callback for some of the requests based on response content.

For more information see [Spider Middleware](#).

### 6.1.4 Event-driven networking

Scrapy is written with [Twisted](#), a popular event-driven networking framework for Python. Thus, it's implemented using a non-blocking (aka asynchronous) code for concurrency.

For more information about asynchronous programming and Twisted see these links:

- [Introduction to Deferreds in Twisted](#)
- [Twisted - hello, asynchronous programming](#)
- [Twisted Introduction - Krondo](#)

## 6.2 Downloader Middleware

The downloader middleware is a framework of hooks into Scrapy's request/response processing. It's a light, low-level system for globally altering Scrapy's requests and responses.

### 6.2.1 Activating a downloader middleware

To activate a downloader middleware component, add it to the **:setting:'DOWNLOADER\_MIDDLEWARES'** setting, which is a dict whose keys are the middleware class paths and their values are the middleware orders.

Here's an example:

```
DOWNLOADER_MIDDLEWARES = {
    'myproject.middlewares.CustomDownloaderMiddleware': 543,
}
```

The **:setting:'DOWNLOADER\_MIDDLEWARES'** setting is merged with the **:setting:'DOWNLOADER\_MIDDLEWARES\_BASE'** setting defined in Scrapy (and not meant to be overridden) and then sorted by order to get the final sorted list of enabled middlewares: the first middleware is the one closer to the engine and the last is the one closer to the downloader. In other words, the `process_request()` method of each middleware will be invoked in increasing middleware order (100, 200, 300, ...) and the `process_response()` method of each middleware will be invoked in decreasing order.

To decide which order to assign to your middleware see the **:setting:'DOWNLOADER\_MIDDLEWARES\_BASE'** setting and pick a value according to where you want to insert the middleware. The order does matter because each middleware performs a different action and your middleware could depend on some previous (or subsequent) middleware being applied.

If you want to disable a built-in middleware (the ones defined in `:setting:'DOWNLOADER_MIDDLEWARES_BASE'` and enabled by default) you must define it in your project's `:setting:'DOWNLOADER_MIDDLEWARES'` setting and assign `None` as its value. For example, if you want to disable the user-agent middleware:

```
DOWNLOADER_MIDDLEWARES = {
    'myproject.middlewares.CustomDownloaderMiddleware': 543,
    'scrapy.downloadermiddlewares.useragent.UserAgentMiddleware': None,
}
```

Finally, keep in mind that some middlewares may need to be enabled through a particular setting. See each middleware documentation for more info.

## 6.2.2 Writing your own downloader middleware

Each middleware component is a Python class that defines one or more of the following methods:

**class** scrapy.downloadermiddlewares.DownloaderMiddleware

---

**Note:** Any of the downloader middleware methods may also return a deferred.

---

**process\_request** (*request*, *spider*)

This method is called for each request that goes through the download middleware.

*process\_request()* should either: return `None`, return a *Response* object, return a *Request* object, or raise *IgnoreRequest*.

If it returns `None`, Scrapy will continue processing this request, executing all other middlewares until, finally, the appropriate downloader handler is called the request performed (and its response downloaded).

If it returns a *Response* object, Scrapy won't bother calling *any* other *process\_request()* or *process\_exception()* methods, or the appropriate download function; it'll return that response. The *process\_response()* methods of installed middleware is always called on every response.

If it returns a *Request* object, Scrapy will stop calling *process\_request* methods and reschedule the returned request. Once the newly returned request is performed, the appropriate middleware chain will be called on the downloaded response.

If it raises an *IgnoreRequest* exception, the *process\_exception()* methods of installed downloader middleware will be called. If none of them handle the exception, the *errback* function of the request (*Request.errback*) is called. If no code handles the raised exception, it is ignored and not logged (unlike other exceptions).

### Parameters

- **request** (*Request* object) – the request being processed
- **spider** (*Spider* object) – the spider for which this request is intended

**process\_response** (*request*, *response*, *spider*)

*process\_response()* should either: return a *Response* object, return a *Request* object or raise a *IgnoreRequest* exception.

If it returns a *Response* (it could be the same given response, or a brand-new one), that response will continue to be processed with the *process\_response()* of the next middleware in the chain.

If it returns a `Request` object, the middleware chain is halted and the returned request is rescheduled to be downloaded in the future. This is the same behavior as if a request is returned from `process_request()`.

If it raises an `IgnoreRequest` exception, the errback function of the request (`Request.errback`) is called. If no code handles the raised exception, it is ignored and not logged (unlike other exceptions).

#### Parameters

- **request** (is a `Request` object) – the request that originated the response
- **response** (`Response` object) – the response being processed
- **spider** (`Spider` object) – the spider for which this response is intended

#### `process_exception(request, exception, spider)`

Scrapy calls `process_exception()` when a download handler or a `process_request()` (from a downloader middleware) raises an exception (including an `IgnoreRequest` exception)

`process_exception()` should return: either `None`, a `Response` object, or a `Request` object.

If it returns `None`, Scrapy will continue processing this exception, executing any other `process_exception()` methods of installed middleware, until no middleware is left and the default exception handling kicks in.

If it returns a `Response` object, the `process_response()` method chain of installed middleware is started, and Scrapy won't bother calling any other `process_exception()` methods of middleware.

If it returns a `Request` object, the returned request is rescheduled to be downloaded in the future. This stops the execution of `process_exception()` methods of the middleware the same as returning a response would.

#### Parameters

- **request** (is a `Request` object) – the request that generated the exception
- **exception** (an `Exception` object) – the raised exception
- **spider** (`Spider` object) – the spider for which this request is intended

#### `from_crawler(cls, crawler)`

If present, this classmethod is called to create a middleware instance from a `Crawler`. It must return a new instance of the middleware. `Crawler` object provides access to all Scrapy core components like settings and signals; it is a way for middleware to access them and hook its functionality into Scrapy.

**Parameters** **crawler** (`Crawler` object) – crawler that uses this middleware

## 6.2.3 Built-in downloader middleware reference

This page describes all downloader middleware components that come with Scrapy. For information on how to use them and how to write your own downloader middleware, see the [downloader middleware usage guide](#).

For a list of the components enabled by default (and their orders) see the **:setting:'DOWNLOADER\_MIDDLEWARES\_BASE'** setting.

### CookiesMiddleware

#### `class scrapy.downloadermiddlewares.cookies.CookiesMiddleware`

This middleware enables working with sites that require cookies, such as those that use sessions. It keeps track of cookies sent by web servers, and send them back on subsequent requests (from that spider), just like web browsers do.



The following settings can be used to configure the cookie middleware:

- **:setting:‘COOKIES\_ENABLED‘**
- **:setting:‘COOKIES\_DEBUG‘**

## Multiple cookie sessions per spider

New in version 0.15.

There is support for keeping multiple cookie sessions per spider by using the **:reqmeta:‘cookiejar‘** Request meta key. By default it uses a single cookie jar (session), but you can pass an identifier to use different ones.

For example:

```
for i, url in enumerate(urls):
    yield scrapy.Request(url, meta={'cookiejar': i},
                        callback=self.parse_page)
```

Keep in mind that the **:reqmeta:‘cookiejar‘** meta key is not “sticky”. You need to keep passing it along on subsequent requests. For example:

```
def parse_page(self, response):
    # do some processing
    return scrapy.Request("http://www.example.com/otherpage",
                        meta={'cookiejar': response.meta['cookiejar']},
                        callback=self.parse_other_page)
```

## COOKIES\_ENABLED

Default: True

Whether to enable the cookies middleware. If disabled, no cookies will be sent to web servers.

Notice that despite the value of **:setting:‘COOKIES\_ENABLED‘** setting if `Request. :reqmeta:‘meta[‘dont_merge_cookies’] <dont_merge_cookies>‘` evaluates to True the request cookies will **not** be sent to the web server and received cookies in `Response` will **not** be merged with the existing cookies.

For more detailed information see the `cookies` parameter in `Request`.

## COOKIES\_DEBUG

Default: False

If enabled, Scrapy will log all cookies sent in requests (ie. `Cookie` header) and all cookies received in responses (ie. `Set-Cookie` header).

Here’s an example of a log with **:setting:‘COOKIES\_DEBUG‘** enabled:

```
2011-04-06 14:35:10-0300 [scrapy.core.engine] INFO: Spider opened
2011-04-06 14:35:10-0300 [scrapy.downloadermiddlewares.cookies] DEBUG: Sending_
→cookies to: <GET http://www.diningcity.com/netherlands/index.html>
      Cookie: clientlanguage_nl=en_EN
2011-04-06 14:35:14-0300 [scrapy.downloadermiddlewares.cookies] DEBUG: Received_
→cookies from: <200 http://www.diningcity.com/netherlands/index.html>
      Set-Cookie: JSESSIONID=B~FA4DC0C496C8762AE4F1A620EAB34F38; Path=/
```

(continues on next page)

(continued from previous page)

```
Set-Cookie: ip_isocode=US
Set-Cookie: clientlanguage_nl=en_EN; Expires=Thu, 07-Apr-2011 21:21:34 GMT;␣
↪Path=/
2011-04-06 14:49:50-0300 [scrapy.core.engine] DEBUG: Crawled (200) <GET http://www.
↪diningcity.com/netherlands/index.html> (referer: None)
[...]
```

## DefaultHeadersMiddleware

**class** scrapy.downloadermiddlewares.defaultheaders.**DefaultHeadersMiddleware**  
This middleware sets all default requests headers specified in the **:setting:‘DEFAULT\_REQUEST\_HEADERS’** setting.

## DownloadTimeoutMiddleware

**class** scrapy.downloadermiddlewares.downloadtimeout.**DownloadTimeoutMiddleware**  
This middleware sets the download timeout for requests specified in the **:setting:‘DOWNLOAD\_TIMEOUT’** setting or download\_timeout spider attribute.

---

**Note:** You can also set download timeout per-request using **:reqmeta:‘download\_timeout’** Request.meta key; this is supported even when DownloadTimeoutMiddleware is disabled.

---

## HttpAuthMiddleware

**class** scrapy.downloadermiddlewares.httputh.**HttpAuthMiddleware**  
This middleware authenticates all requests generated from certain spiders using **Basic access authentication** (aka. HTTP auth).

To enable HTTP authentication from certain spiders, set the http\_user and http\_pass attributes of those spiders.

Example:

```
from scrapy.spiders import CrawlSpider

class SomeIntranetSiteSpider(CrawlSpider):

    http_user = 'someuser'
    http_pass = 'somepass'
    name = 'intranet.example.com'

    # .. rest of the spider code omitted ...
```

## HttpCacheMiddleware

**class** scrapy.downloadermiddlewares.httppcache.**HttpCacheMiddleware**  
This middleware provides low-level cache to all HTTP requests and responses. It has to be combined with a cache storage backend as well as a cache policy.

Scrapy ships with three HTTP cache storage backends:

- *Filesystem storage backend (default)*
- *DBM storage backend*
- *LevelDB storage backend*

You can change the HTTP cache storage backend with the **:setting:‘HTTPCACHE\_STORAGE’** setting. Or you can also implement your own storage backend.

Scrapy ships with two HTTP cache policies:

- *RFC2616 policy*
- *Dummy policy (default)*

You can change the HTTP cache policy with the **:setting:‘HTTPCACHE\_POLICY’** setting. Or you can also implement your own policy.

You can also avoid caching a response on every policy using **:reqmeta:‘dont\_cache’** meta key equals *True*.

### Dummy policy (default)

This policy has no awareness of any HTTP Cache-Control directives. Every request and its corresponding response are cached. When the same request is seen again, the response is returned without transferring anything from the Internet.

The Dummy policy is useful for testing spiders faster (without having to wait for downloads every time) and for trying your spider offline, when an Internet connection is not available. The goal is to be able to “replay” a spider run *exactly as it ran before*.

In order to use this policy, set:

- **:setting:‘HTTPCACHE\_POLICY’** to `scrapy.extensions.httppcache.DummyPolicy`

### RFC2616 policy

This policy provides a RFC2616 compliant HTTP cache, i.e. with HTTP Cache-Control awareness, aimed at production and used in continuous runs to avoid downloading unmodified data (to save bandwidth and speed up crawls).

what is implemented:

- Do not attempt to store responses/requests with *no-store* cache-control directive set
- Do not serve responses from cache if *no-cache* cache-control directive is set even for fresh responses
- Compute freshness lifetime from *max-age* cache-control directive
- Compute freshness lifetime from *Expires* response header
- Compute freshness lifetime from *Last-Modified* response header (heuristic used by Firefox)
- Compute current age from *Age* response header
- Compute current age from *Date* header
- Revalidate stale responses based on *Last-Modified* response header
- Revalidate stale responses based on *ETag* response header
- Set *Date* header for any received response missing it
- Support *max-stale* cache-control directive in requests

This allows spiders to be configured with the full RFC2616 cache policy, but avoid revalidation on a request-by-request basis, while remaining conformant with the HTTP spec.

Example:

Add *Cache-Control: max-stale=600* to Request headers to accept responses that have exceeded their expiration time by no more than 600 seconds.

See also: RFC2616, 14.9.3

what is missing:

- *Pragma: no-cache* support <https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.9.1>
- *Vary* header support <https://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13.6>
- Invalidation after updates or deletes <https://www.w3.org/Protocols/rfc2616/rfc2616-sec13.html#sec13.10>
- ... probably others ..

In order to use this policy, set:

- **:setting:'HTTPCACHE\_POLICY'** to `scrapy.extensions.httppcache.RFC2616Policy`

### Filesystem storage backend (default)

File system storage backend is available for the HTTP cache middleware.

In order to use this storage backend, set:

- **:setting:'HTTPCACHE\_STORAGE'** to `scrapy.extensions.httppcache.FileSystemCacheStorage`

Each request/response pair is stored in a different directory containing the following files:

- `request_body` - the plain request body
- `request_headers` - the request headers (in raw HTTP format)
- `response_body` - the plain response body
- `response_headers` - the request headers (in raw HTTP format)
- `meta` - some metadata of this cache resource in Python `repr()` format (grep-friendly format)
- `pickled_meta` - the same metadata in `meta` but pickled for more efficient deserialization

The directory name is made from the request fingerprint (see `scrapy.utils.request.fingerprint`), and one level of subdirectories is used to avoid creating too many files into the same directory (which is inefficient in many file systems). An example directory could be:

`/path/to/cache/dir/example.com/72/72811f648e718090f041317756c03adb0ada46c7`

### DBM storage backend

New in version 0.13.

A **DBM** storage backend is also available for the HTTP cache middleware.

By default, it uses the `anydbm` module, but you can change it with the **:setting:'HTTPCACHE\_DBM\_MODULE'** setting.

In order to use this storage backend, set:

- **:setting:'HTTPCACHE\_STORAGE'** to `scrapy.extensions.httppcache.DbmCacheStorage`

## LevelDB storage backend

New in version 0.23.

A [LevelDB](#) storage backend is also available for the HTTP cache middleware.

This backend is not recommended for development because only one process can access LevelDB databases at the same time, so you can't run a crawl and open the scrapy shell in parallel for the same spider.

In order to use this storage backend:

- set **:setting:'HTTPCACHE\_STORAGE'** to `scrapy.extensions.httpcache.LevelDbCacheStorage`
- install [LevelDB python bindings](#) like `pip install leveldb`

## HTTPCache middleware settings

The [HttpCacheMiddleware](#) can be configured through the following settings:

### HTTPCACHE\_ENABLED

New in version 0.11.

Default: `False`

Whether the HTTP cache will be enabled.

Changed in version 0.11: Before 0.11, **:setting:'HTTPCACHE\_DIR'** was used to enable cache.

### HTTPCACHE\_EXPIRATION\_SECS

Default: `0`

Expiration time for cached requests, in seconds.

Cached requests older than this time will be re-downloaded. If zero, cached requests will never expire.

Changed in version 0.11: Before 0.11, zero meant cached requests always expire.

### HTTPCACHE\_DIR

Default: `'httpcache'`

The directory to use for storing the (low-level) HTTP cache. If empty, the HTTP cache will be disabled. If a relative path is given, is taken relative to the project data dir. For more info see: [Default structure of Scrapy projects](#).

### HTTPCACHE\_IGNORE\_HTTP\_CODES

New in version 0.10.

Default: `[]`

Don't cache response with these HTTP codes.

## HTTPCACHE\_IGNORE\_MISSING

Default: `False`

If enabled, requests not found in the cache will be ignored instead of downloaded.

## HTTPCACHE\_IGNORE\_SCHEMES

New in version 0.10.

Default: `['file']`

Don't cache responses with these URI schemes.

## HTTPCACHE\_STORAGE

Default: `'scrapy.extensions.httpcache.FilesystemCacheStorage'`

The class which implements the cache storage backend.

## HTTPCACHE\_DBM\_MODULE

New in version 0.13.

Default: `'anydbm'`

The database module to use in the *DBM storage backend*. This setting is specific to the DBM backend.

## HTTPCACHE\_POLICY

New in version 0.18.

Default: `'scrapy.extensions.httpcache.DummyPolicy'`

The class which implements the cache policy.

## HTTPCACHE\_GZIP

New in version 1.0.

Default: `False`

If enabled, will compress all cached data with gzip. This setting is specific to the Filesystem backend.

## HTTPCACHE\_ALWAYS\_STORE

New in version 1.1.

Default: `False`

If enabled, will cache pages unconditionally.

A spider may wish to have all responses available in the cache, for future use with *Cache-Control: max-stale*, for instance. The DummyPolicy caches all responses but never revalidates them, and sometimes a more nuanced policy is desirable.

This setting still respects *Cache-Control: no-store* directives in responses. If you don't want that, filter *no-store* out of the Cache-Control headers in responses you feed to the cache middleware.

## HTTPCACHE\_IGNORE\_RESPONSE\_CACHE\_CONTROLS

New in version 1.1.

Default: []

List of Cache-Control directives in responses to be ignored.

Sites often set “no-store”, “no-cache”, “must-revalidate”, etc., but get upset at the traffic a spider can generate if it respects those directives. This allows to selectively ignore Cache-Control directives that are known to be unimportant for the sites being crawled.

We assume that the spider will not issue Cache-Control directives in requests unless it actually needs them, so directives in requests are not filtered.

## HttpCompressionMiddleware

**class** scrapy.downloadermiddlewares.httpcompression.HttpCompressionMiddleware

This middleware allows compressed (gzip, deflate) traffic to be sent/received from web sites.

This middleware also supports decoding [brotli-compressed](#) responses, provided [brotlipy](#) is installed.

## HttpCompressionMiddleware Settings

### COMPRESSION\_ENABLED

Default: True

Whether the Compression middleware will be enabled.

## HttpProxyMiddleware

New in version 0.8.

**class** scrapy.downloadermiddlewares.httpproxy.HttpProxyMiddleware

This middleware sets the HTTP proxy to use for requests, by setting the `proxy` meta value for *Request* objects.

Like the Python standard library modules [urllib](#) and [urllib2](#), it obeys the following environment variables:

- `http_proxy`
- `https_proxy`
- `no_proxy`

You can also set the meta key `proxy` per-request, to a value like `http://some_proxy_server:port` or `http://username:password@some_proxy_server:port`. Keep in mind this value will take precedence over `http_proxy/https_proxy` environment variables, and it will also ignore `no_proxy` environment variable.

## RedirectMiddleware

**class** scrapy.downloadermiddlewares.redirect.**RedirectMiddleware**

This middleware handles redirection of requests based on response status.

The urls which the request goes through (while being redirected) can be found in the `redirect_urls` *Request*.*meta* key.

The *RedirectMiddleware* can be configured through the following settings (see the settings documentation for more info):

- **:setting:‘REDIRECT\_ENABLED‘**
- **:setting:‘REDIRECT\_MAX\_TIMES‘**

If *Request*.*meta* has `dont_redirect` key set to `True`, the request will be ignored by this middleware.

If you want to handle some redirect status codes in your spider, you can specify these in the `handle_httpstatus_list` spider attribute.

For example, if you want the redirect middleware to ignore 301 and 302 responses (and pass them through to your spider) you can do this:

```
class MySpider(CrawlSpider):
    handle_httpstatus_list = [301, 302]
```

The `handle_httpstatus_list` key of *Request*.*meta* can also be used to specify which response codes to allow on a per-request basis. You can also set the meta key `handle_httpstatus_all` to `True` if you want to allow any response code for a request.

## RedirectMiddleware settings

### REDIRECT\_ENABLED

New in version 0.13.

Default: `True`

Whether the Redirect middleware will be enabled.

### REDIRECT\_MAX\_TIMES

Default: `20`

The maximum number of redirections that will be followed for a single request.

## MetaRefreshMiddleware

**class** scrapy.downloadermiddlewares.redirect.**MetaRefreshMiddleware**

This middleware handles redirection of requests based on meta-refresh html tag.

The *MetaRefreshMiddleware* can be configured through the following settings (see the settings documentation for more info):

- **:setting:‘METAREFRESH\_ENABLED‘**
- **:setting:‘METAREFRESH\_MAXDELAY‘**



This middleware obey **:setting:'REDIRECT\_MAX\_TIMES'** setting, **:reqmeta:'dont\_redirect'** and **:reqmeta:'redirect\_urls'** request meta keys as described for *RedirectMiddleware*

## MetaRefreshMiddleware settings

### METAREFRESH\_ENABLED

New in version 0.17.

Default: True

Whether the Meta Refresh middleware will be enabled.

### METAREFRESH\_MAXDELAY

Default: 100

The maximum meta-refresh delay (in seconds) to follow the redirection. Some sites use meta-refresh for redirecting to a session expired page, so we restrict automatic redirection to the maximum delay.

## RetryMiddleware

**class** scrapy.downloadermiddlewares.retry.RetryMiddleware

A middleware to retry failed requests that are potentially caused by temporary problems such as a connection timeout or HTTP 500 error.

Failed pages are collected on the scraping process and rescheduled at the end, once the spider has finished crawling all regular (non failed) pages. Once there are no more failed pages to retry, this middleware sends a signal (retry\_complete), so other extensions could connect to that signal.

The *RetryMiddleware* can be configured through the following settings (see the settings documentation for more info):

- **:setting:'RETRY\_ENABLED'**
- **:setting:'RETRY\_TIMES'**
- **:setting:'RETRY\_HTTP\_CODES'**

If *Request.meta* has dont\_retry key set to True, the request will be ignored by this middleware.

## RetryMiddleware Settings

### RETRY\_ENABLED

New in version 0.13.

Default: True

Whether the Retry middleware will be enabled.

## RETRY\_TIMES

Default: 2

Maximum number of times to retry, in addition to the first download.

Maximum number of retries can also be specified per-request using `:reqmeta:'max_retry_times'` attribute of `Request.meta`. When initialized, the `:reqmeta:'max_retry_times'` meta key takes higher precedence over the `:setting:'RETRY_TIMES'` setting.

## RETRY\_HTTP\_CODES

Default: [500, 502, 503, 504, 522, 524, 408]

Which HTTP response codes to retry. Other errors (DNS lookup issues, connections lost, etc) are always retried.

In some cases you may want to add 400 to `:setting:'RETRY_HTTP_CODES'` because it is a common code used to indicate server overload. It is not included by default because HTTP specs say so.

## RobotsTxtMiddleware

**class** scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware

This middleware filters out requests forbidden by the robots.txt exclusion standard.

To make sure Scrapy respects robots.txt make sure the middleware is enabled and the `:setting:'ROBOTSTXT_OBEY'` setting is enabled.

If `Request.meta` has `dont_obey_robotstxt` key set to True the request will be ignored by this middleware even if `:setting:'ROBOTSTXT_OBEY'` is enabled.

## DownloaderStats

**class** scrapy.downloadermiddlewares.stats.DownloaderStats

Middleware that stores stats of all requests, responses and exceptions that pass through it.

To use this middleware you must enable the `:setting:'DOWNLOADER_STATS'` setting.

## UserAgentMiddleware

**class** scrapy.downloadermiddlewares.useragent.UserAgentMiddleware

Middleware that allows spiders to override the default user agent.

In order for a spider to override the default user agent, its `user_agent` attribute must be set.

## AjaxCrawlMiddleware

**class** scrapy.downloadermiddlewares.ajaxcrawl.AjaxCrawlMiddleware

Middleware that finds 'AJAX crawlable' page variants based on meta-fragment html tag. See <https://developers.google.com/webmasters/ajax-crawling/docs/getting-started> for more info.

---

**Note:** Scrapy finds 'AJAX crawlable' pages for URLs like `'http://example.com/!#foo=bar'` even without this middleware. AjaxCrawlMiddleware is necessary when URL doesn't contain `'!#'`. This is often a case for 'index' or 'main' website pages.

---

## AjaxCrawlMiddleware Settings

### AJAXCRAWL\_ENABLED

New in version 0.21.

Default: `False`

Whether the AjaxCrawlMiddleware will be enabled. You may want to enable it for *broad crawls*.

## HttpProxyMiddleware settings

### HTTPPROXY\_ENABLED

Default: `True`

Whether or not to enable the HttpProxyMiddleware.

### HTTPPROXY\_AUTH\_ENCODING

Default: `"latin-1"`

The default encoding for proxy authentication on HttpProxyMiddleware.

## 6.3 Spider Middleware

The spider middleware is a framework of hooks into Scrapy's spider processing mechanism where you can plug custom functionality to process the responses that are sent to *Spiders* for processing and to process the requests and items that are generated from spiders.

### 6.3.1 Activating a spider middleware

To activate a spider middleware component, add it to the **:setting:'SPIDER\_MIDDLEWARES'** setting, which is a dict whose keys are the middleware class path and their values are the middleware orders.

Here's an example:

```
SPIDER_MIDDLEWARES = {
    'myproject.middlewares.CustomSpiderMiddleware': 543,
}
```

The **:setting:'SPIDER\_MIDDLEWARES'** setting is merged with the **:setting:'SPIDER\_MIDDLEWARES\_BASE'** setting defined in Scrapy (and not meant to be overridden) and then sorted by order to get the final sorted list of enabled middlewares: the first middleware is the one closer to the engine and the last is the one closer to the spider. In other words, the `process_spider_input()` method of each middleware will be invoked in increasing middleware order (100, 200, 300, ...), and the `process_spider_output()` method of each middleware will be invoked in decreasing order.

To decide which order to assign to your middleware see the **:setting:'SPIDER\_MIDDLEWARES\_BASE'** setting and pick a value according to where you want to insert the middleware. The order does matter because each middleware performs a different action and your middleware could depend on some previous (or subsequent) middleware being applied.

If you want to disable a builtin middleware (the ones defined in `:setting:'SPIDER_MIDDLEWARES_BASE'`, and enabled by default) you must define it in your project `:setting:'SPIDER_MIDDLEWARES'` setting and assign `None` as its value. For example, if you want to disable the off-site middleware:

```
SPIDER_MIDDLEWARES = {
    'myproject.middlewares.CustomSpiderMiddleware': 543,
    'scrapy.spidermiddlewares.offsite.OffsiteMiddleware': None,
}
```

Finally, keep in mind that some middlewares may need to be enabled through a particular setting. See each middleware documentation for more info.

## 6.3.2 Writing your own spider middleware

Each middleware component is a Python class that defines one or more of the following methods:

**class** scrapy.spidermiddlewares.SpiderMiddleware

**process\_spider\_input** (*response*, *spider*)

This method is called for each response that goes through the spider middleware and into the spider, for processing.

`process_spider_input()` should return `None` or raise an exception.

If it returns `None`, Scrapy will continue processing this response, executing all other middlewares until, finally, the response is handed to the spider for processing.

If it raises an exception, Scrapy won't bother calling any other spider middleware `process_spider_input()` and will call the request errback. The output of the errback is chained back in the other direction for `process_spider_output()` to process it, or `process_spider_exception()` if it raised an exception.

### Parameters

- **response** (*Response* object) – the response being processed
- **spider** (*Spider* object) – the spider for which this response is intended

**process\_spider\_output** (*response*, *result*, *spider*)

This method is called with the results returned from the Spider, after it has processed the response.

`process_spider_output()` must return an iterable of *Request*, dict or *Item* objects.

### Parameters

- **response** (*Response* object) – the response which generated this output from the spider
- **result** (an iterable of *Request*, dict or *Item* objects) – the result returned by the spider
- **spider** (*Spider* object) – the spider whose result is being processed

**process\_spider\_exception** (*response*, *exception*, *spider*)

This method is called when a spider or `process_spider_input()` method (from other spider middleware) raises an exception.

`process_spider_exception()` should return either `None` or an iterable of *Request*, dict or *Item* objects.

If it returns `None`, Scrapy will continue processing this exception, executing any other `process_spider_exception()` in the following middleware components, until no middleware components are left and the exception reaches the engine (where it's logged and discarded).

If it returns an iterable the `process_spider_output()` pipeline kicks in, and no other `process_spider_exception()` will be called.

#### Parameters

- **response** (*Response* object) – the response being processed when the exception was raised
- **exception** (*Exception* object) – the exception raised
- **spider** (*Spider* object) – the spider which raised the exception

**process\_start\_requests** (*start\_requests*, *spider*)

New in version 0.15.

This method is called with the start requests of the spider, and works similarly to the `process_spider_output()` method, except that it doesn't have a response associated and must return only requests (not items).

It receives an iterable (in the `start_requests` parameter) and must return another iterable of *Request* objects.

---

**Note:** When implementing this method in your spider middleware, you should always return an iterable (that follows the input one) and not consume all `start_requests` iterator because it can be very large (or even unbounded) and cause a memory overflow. The Scrapy engine is designed to pull start requests while it has capacity to process them, so the start requests iterator can be effectively endless where there is some other condition for stopping the spider (like a time limit or item/page count).

---

#### Parameters

- **start\_requests** (an iterable of *Request*) – the start requests
- **spider** (*Spider* object) – the spider to whom the start requests belong

**from\_crawler** (*cls*, *crawler*)

If present, this classmethod is called to create a middleware instance from a *Crawler*. It must return a new instance of the middleware. Crawler object provides access to all Scrapy core components like settings and signals; it is a way for middleware to access them and hook its functionality into Scrapy.

**Parameters** **crawler** (*Crawler* object) – crawler that uses this middleware

## 6.3.3 Built-in spider middleware reference

This page describes all spider middleware components that come with Scrapy. For information on how to use them and how to write your own spider middleware, see the [spider middleware usage guide](#).

For a list of the components enabled by default (and their orders) see the **setting: 'SPIDER\_MIDDLEWARES\_BASE'** setting.

### DepthMiddleware

**class** scrapy.spidermiddlewares.depth.DepthMiddleware

DepthMiddleware is used for tracking the depth of each Request inside the site being scraped. It works by

setting `request.meta['depth'] = 0` whenever there is no value previously set (usually just the first Request) and incrementing it by 1 otherwise.

It can be used to limit the maximum depth to scrape, control Request priority based on their depth, and things like that.

The `DepthMiddleware` can be configured through the following settings (see the settings documentation for more info):

- **:setting:‘DEPTH\_LIMIT’** - The maximum depth that will be allowed to crawl for any site. If zero, no limit will be imposed.
- **:setting:‘DEPTH\_STATS\_VERBOSE’** - Whether to collect the number of requests for each depth.
- **:setting:‘DEPTH\_PRIORITY’** - Whether to prioritize the requests based on their depth.

## HttpErrorMiddleware

**class** scrapy.spidermiddlewares.httperror.HttpErrorMiddleware

Filter out unsuccessful (erroneous) HTTP responses so that spiders don’t have to deal with them, which (most of the time) imposes an overhead, consumes more resources, and makes the spider logic more complex.

According to the [HTTP standard](#), successful responses are those whose status codes are in the 200-300 range.

If you still want to process response codes outside that range, you can specify which response codes the spider is able to handle using the `handle_httpstatus_list` spider attribute or **:setting:‘HTTPELLOWED\_CODES’** setting.

For example, if you want your spider to handle 404 responses you can do this:

```
class MySpider(CrawlSpider):
    handle_httpstatus_list = [404]
```

The `handle_httpstatus_list` key of `Request.meta` can also be used to specify which response codes to allow on a per-request basis. You can also set the meta key `handle_httpstatus_all` to `True` if you want to allow any response code for a request.

Keep in mind, however, that it’s usually a bad idea to handle non-200 responses, unless you really know what you’re doing.

For more information see: [HTTP Status Code Definitions](#).

## HttpErrorMiddleware settings

### HTTPELLOWED\_CODES

Default: []

Pass all responses with non-200 status codes contained in this list.

### HTTPELLOW\_ALL

Default: False

Pass all responses, regardless of its status code.

## OffsiteMiddleware

**class** scrapy.spidermiddlewares.offsite.**OffsiteMiddleware**

Filters out Requests for URLs outside the domains covered by the spider.

This middleware filters out every request whose host names aren't in the spider's `allowed_domains` attribute. All subdomains of any domain in the list are also allowed. E.g. the rule `www.example.org` will also allow `bob.www.example.org` but not `www2.example.com` nor `example.com`.

When your spider returns a request for a domain not belonging to those covered by the spider, this middleware will log a debug message similar to this one:

```
DEBUG: Filtered offsite request to 'www.othersite.com': <GET http://www.othersite.
↪com/some/page.html>
```

To avoid filling the log with too much noise, it will only print one of these messages for each new domain filtered. So, for example, if another request for `www.othersite.com` is filtered, no log message will be printed. But if a request for `someothersite.com` is filtered, a message will be printed (but only for the first request filtered).

If the spider doesn't define an `allowed_domains` attribute, or the attribute is empty, the offsite middleware will allow all requests.

If the request has the `dont_filter` attribute set, the offsite middleware will allow the request even if its domain is not listed in allowed domains.

## RefererMiddleware

**class** scrapy.spidermiddlewares.referer.**RefererMiddleware**

Populates Request Referer header, based on the URL of the Response which generated it.

### RefererMiddleware settings

#### REFERER\_ENABLED

New in version 0.15.

Default: True

Whether to enable referer middleware.

#### REFERRER\_POLICY

New in version 1.4.

Default: 'scrapy.spidermiddlewares.referer.DefaultReferrerPolicy'

[Referrer Policy](#) to apply when populating Request "Referer" header.

---

**Note:** You can also set the Referrer Policy per request, using the special "referrer\_policy" [Request.meta](#) key, with the same acceptable values as for the `REFERRER_POLICY` setting.

---

## Acceptable values for REFERRER\_POLICY

- either a path to a `scrapy.spidermiddlewares.referer.RefererPolicy` subclass — a custom policy or one of the built-in ones (see classes below),
- or one of the standard W3C-defined string values,
- or the special `"scrapy-default"`.

String value	Class name (as a string)
<code>"scrapy-default"</code> (default)	<code>scrapy.spidermiddlewares.referer.DefaultRefererPolicy</code>
<code>"no-referrer"</code>	<code>scrapy.spidermiddlewares.referer.NoRefererPolicy</code>
<code>"no-referrer-when-downgrade"</code>	<code>scrapy.spidermiddlewares.referer.NoRefererWhenDowngradePolicy</code>
<code>"same-origin"</code>	<code>scrapy.spidermiddlewares.referer.SameOriginPolicy</code>
<code>"origin"</code>	<code>scrapy.spidermiddlewares.referer.OriginPolicy</code>
<code>"strict-origin"</code>	<code>scrapy.spidermiddlewares.referer.StrictOriginPolicy</code>
<code>"origin-when-cross-origin"</code>	<code>scrapy.spidermiddlewares.referer.OriginWhenCrossOriginPolicy</code>
<code>"strict-origin-when-cross-origin"</code>	<code>scrapy.spidermiddlewares.referer.StrictOriginWhenCrossOriginPolicy</code>
<code>"unsafe-url"</code>	<code>scrapy.spidermiddlewares.referer.UnsafeUrlPolicy</code>

**Warning:** Scrapy’s default referrer policy — just like `"no-referrer-when-downgrade"`, the W3C-recommended value for browsers — will send a non-empty `"Referer"` header from any `http(s)://` to any `https://` URL, even if the domain is different.

`"same-origin"` may be a better choice if you want to remove referrer information for cross-domain requests.

---

**Note:** `"no-referrer-when-downgrade"` policy is the W3C-recommended default, and is used by major web browsers. However, it is NOT Scrapy’s default referrer policy (see `DefaultRefererPolicy`).

---

**Warning:** `"unsafe-url"` policy is NOT recommended.

## UrlLengthMiddleware

**class** `scrapy.spidermiddlewares.urllength.UrlLengthMiddleware`

Filters out requests with URLs longer than `URLLENGTH_LIMIT`

The `UrlLengthMiddleware` can be configured through the following settings (see the settings documentation for more info):

- **setting:‘URLLENGTH\_LIMIT’** - The maximum URL length to allow for crawled URLs.

## 6.4 Extensions

The extensions framework provides a mechanism for inserting your own custom functionality into Scrapy.



Extensions are just regular classes that are instantiated at Scrapy startup, when extensions are initialized.

### 6.4.1 Extension settings

Extensions use the *Scrapy settings* to manage their settings, just like any other Scrapy code.

It is customary for extensions to prefix their settings with their own name, to avoid collision with existing (and future) extensions. For example, a hypothetical extension to handle [Google Sitemaps](#) would use settings like `GOOGLE-SITEMAP_ENABLED`, `GOOGLESITEMAP_DEPTH`, and so on.

### 6.4.2 Loading & activating extensions

Extensions are loaded and activated at startup by instantiating a single instance of the extension class. Therefore, all the extension initialization code must be performed in the class constructor (`__init__` method).

To make an extension available, add it to the `:setting:'EXTENSIONS'` setting in your Scrapy settings. In `:setting:'EXTENSIONS'`, each extension is represented by a string: the full Python path to the extension's class name. For example:

```
EXTENSIONS = {
    'scrapy.extensions.corestats.CoreStats': 500,
    'scrapy.extensions.telnet.TelnetConsole': 500,
}
```

As you can see, the `:setting:'EXTENSIONS'` setting is a dict where the keys are the extension paths, and their values are the orders, which define the extension *loading* order. The `:setting:'EXTENSIONS'` setting is merged with the `:setting:'EXTENSIONS_BASE'` setting defined in Scrapy (and not meant to be overridden) and then sorted by order to get the final sorted list of enabled extensions.

As extensions typically do not depend on each other, their loading order is irrelevant in most cases. This is why the `:setting:'EXTENSIONS_BASE'` setting defines all extensions with the same order (0). However, this feature can be exploited if you need to add an extension which depends on other extensions already loaded.

### 6.4.3 Available, enabled and disabled extensions

Not all available extensions will be enabled. Some of them usually depend on a particular setting. For example, the HTTP Cache extension is available by default but disabled unless the `:setting:'HTTPCACHE_ENABLED'` setting is set.

### 6.4.4 Disabling an extension

In order to disable an extension that comes enabled by default (ie. those included in the `:setting:'EXTENSIONS_BASE'` setting) you must set its order to `None`. For example:

```
EXTENSIONS = {
    'scrapy.extensions.corestats.CoreStats': None,
}
```

## 6.4.5 Writing your own extension

Each extension is a Python class. The main entry point for a Scrapy extension (this also includes middlewares and pipelines) is the `from_crawler` class method which receives a `Crawler` instance. Through the `Crawler` object you can access settings, signals, stats, and also control the crawling behaviour.

Typically, extensions connect to *signals* and perform tasks triggered by them.

Finally, if the `from_crawler` method raises the *NotConfigured* exception, the extension will be disabled. Otherwise, the extension will be enabled.

### Sample extension

Here we will implement a simple extension to illustrate the concepts described in the previous section. This extension will log a message every time:

- a spider is opened
- a spider is closed
- a specific number of items are scraped

The extension will be enabled through the `MYEXT_ENABLED` setting and the number of items will be specified through the `MYEXT_ITEMCOUNT` setting.

Here is the code of such extension:

```
import logging
from scrapy import signals
from scrapy.exceptions import NotConfigured

logger = logging.getLogger(__name__)

class SpiderOpenCloseLogging(object):

    def __init__(self, item_count):
        self.item_count = item_count
        self.items_scraped = 0

    @classmethod
    def from_crawler(cls, crawler):
        # first check if the extension should be enabled and raise
        # NotConfigured otherwise
        if not crawler.settings.getbool('MYEXT_ENABLED'):
            raise NotConfigured

        # get the number of items from settings
        item_count = crawler.settings.getint('MYEXT_ITEMCOUNT', 1000)

        # instantiate the extension object
        ext = cls(item_count)

        # connect the extension object to signals
        crawler.signals.connect(ext.spider_opened, signal=signals.spider_opened)
        crawler.signals.connect(ext.spider_closed, signal=signals.spider_closed)
        crawler.signals.connect(ext.item_scraped, signal=signals.item_scraped)

        # return the extension object
        return ext
```

(continues on next page)

(continued from previous page)

```
def spider_opened(self, spider):
    logger.info("opened spider %s", spider.name)

def spider_closed(self, spider):
    logger.info("closed spider %s", spider.name)

def item_scraped(self, item, spider):
    self.items_scraped += 1
    if self.items_scraped % self.item_count == 0:
        logger.info("scraped %d items", self.items_scraped)
```

## 6.4.6 Built-in extensions reference

### General purpose extensions

#### Log Stats extension

**class** scrapy.extensions.logstats.**LogStats**

Log basic stats like crawled pages and scraped items.

#### Core Stats extension

**class** scrapy.extensions.corestats.**CoreStats**

Enable the collection of core statistics, provided the stats collection is enabled (see *Stats Collection*).

#### Telnet console extension

**class** scrapy.extensions.telnet.**TelnetConsole**

Provides a telnet console for getting into a Python interpreter inside the currently running Scrapy process, which can be very useful for debugging.

The telnet console must be enabled by the **:setting:‘TELNETCONSOLE\_ENABLED’** setting, and the server will listen in the port specified in **:setting:‘TELNETCONSOLE\_PORT’**.

#### Memory usage extension

**class** scrapy.extensions.memusage.**MemoryUsage**

---

**Note:** This extension does not work in Windows.

---

Monitors the memory used by the Scrapy process that runs the spider and:

1. sends a notification e-mail when it exceeds a certain value
2. closes the spider when it exceeds a certain value

The notification e-mails can be triggered when a certain warning value is reached (**:setting:‘MEMUSAGE\_WARNING\_MB‘**) and when the maximum value is reached (**:setting:‘MEMUSAGE\_LIMIT\_MB‘**) which will also cause the spider to be closed and the Scrapy process to be terminated.

This extension is enabled by the **:setting:‘MEMUSAGE\_ENABLED‘** setting and can be configured with the following settings:

- **:setting:‘MEMUSAGE\_LIMIT\_MB‘**
- **:setting:‘MEMUSAGE\_WARNING\_MB‘**
- **:setting:‘MEMUSAGE\_NOTIFY\_MAIL‘**
- **:setting:‘MEMUSAGE\_CHECK\_INTERVAL\_SECONDS‘**

## Memory debugger extension

**class** scrapy.extensions.memdebug.MemoryDebugger

An extension for debugging memory usage. It collects information about:

- objects uncollected by the Python garbage collector
- objects left alive that shouldn't. For more info, see *Debugging memory leaks with trackref*

To enable this extension, turn on the **:setting:‘MEMDEBUG\_ENABLED‘** setting. The info will be stored in the stats.

## Close spider extension

**class** scrapy.extensions.closespider.CloseSpider

Closes a spider automatically when some conditions are met, using a specific closing reason for each condition.

The conditions for closing a spider can be configured through the following settings:

- **:setting:‘CLOSESPIDER\_TIMEOUT‘**
- **:setting:‘CLOSESPIDER\_ITEMCOUNT‘**
- **:setting:‘CLOSESPIDER\_PAGECOUNT‘**
- **:setting:‘CLOSESPIDER\_ERRORCOUNT‘**

### CLOSESPIDER\_TIMEOUT

Default: 0

An integer which specifies a number of seconds. If the spider remains open for more than that number of second, it will be automatically closed with the reason `closespider_timeout`. If zero (or non set), spiders won't be closed by timeout.

### CLOSESPIDER\_ITEMCOUNT

Default: 0

An integer which specifies a number of items. If the spider scrapes more than that amount and those items are passed by the item pipeline, the spider will be closed with the reason `closespider_itemcount`. Requests which are

currently in the downloader queue (up to **:setting:‘CONCURRENT\_REQUESTS’** requests) are still processed. If zero (or non set), spiders won’t be closed by number of passed items.

## CLOSESPIDER\_PAGECOUNT

New in version 0.11.

Default: 0

An integer which specifies the maximum number of responses to crawl. If the spider crawls more than that, the spider will be closed with the reason `closespider_pagecount`. If zero (or non set), spiders won’t be closed by number of crawled responses.

## CLOSESPIDER\_ERRORCOUNT

New in version 0.11.

Default: 0

An integer which specifies the maximum number of errors to receive before closing the spider. If the spider generates more than that number of errors, it will be closed with the reason `closespider_errorcount`. If zero (or non set), spiders won’t be closed by number of errors.

## StatsMailer extension

```
class scrapy.extensions.statmailer.StatsMailer
```

This simple extension can be used to send a notification e-mail every time a domain has finished scraping, including the Scrapy stats collected. The email will be sent to all recipients specified in the **:setting:‘STATSMAILER\_RCPTS’** setting.

## Debugging extensions

### Stack trace dump extension

```
class scrapy.extensions.debug.StackTraceDump
```

Dumps information about the running process when a `SIGQUIT` or `SIGUSR2` signal is received. The information dumped is the following:

1. engine status (using `scrapy.utils.engine.get_engine_status()`)
2. live references (see *Debugging memory leaks with trackref*)
3. stack trace of all threads

After the stack trace and engine status is dumped, the Scrapy process continues running normally.

This extension only works on POSIX-compliant platforms (ie. not Windows), because the `SIGQUIT` and `SIGUSR2` signals are not available on Windows.

There are at least two ways to send Scrapy the `SIGQUIT` signal:

1. By pressing Ctrl-while a Scrapy process is running (Linux only?)
2. By running this command (assuming `<pid>` is the process id of the Scrapy process):

```
kill -QUIT <pid>
```

## Debugger extension

**class** scrapy.extensions.debug.Debugger

Invokes a [Python debugger](#) inside a running Scrapy process when a [SIGUSR2](#) signal is received. After the debugger is exited, the Scrapy process continues running normally.

For more info see *Debugging in Python*.

This extension only works on POSIX-compliant platforms (ie. not Windows).

## 6.5 Core API

New in version 0.15.

This section documents the Scrapy core API, and it's intended for developers of extensions and middlewares.

### 6.5.1 Crawler API

The main entry point to Scrapy API is the [Crawler](#) object, passed to extensions through the `from_crawler` class method. This object provides access to all Scrapy core components, and it's the only way for extensions to access them and hook their functionality into Scrapy.

The Extension Manager is responsible for loading and keeping track of installed extensions and it's configured through the **:setting: 'EXTENSIONS'** setting which contains a dictionary of all available extensions and their order similar to how you *configure the downloader middlewares*.

**class** scrapy.crawler.Crawler(*spidercls, settings*)

The Crawler object must be instantiated with a `scrapy.spiders.Spider` subclass and a `scrapy.settings.Settings` object.

#### settings

The settings manager of this crawler.

This is used by extensions & middlewares to access the Scrapy settings of this crawler.

For an introduction on Scrapy settings see *Settings*.

For the API see `Settings` class.

#### signals

The signals manager of this crawler.

This is used by extensions & middlewares to hook themselves into Scrapy functionality.

For an introduction on signals see *Signals*.

For the API see `SignalManager` class.

#### stats

The stats collector of this crawler.

This is used from extensions & middlewares to record stats of their behaviour, or access stats collected by other extensions.

For an introduction on stats collection see *Stats Collection*.

For the API see [StatsCollector](#) class.

#### **extensions**

The extension manager that keeps track of enabled extensions.

Most extensions won't need to access this attribute.

For an introduction on extensions and a list of available extensions on Scrapy see [Extensions](#).

#### **engine**

The execution engine, which coordinates the core crawling logic between the scheduler, downloader and spiders.

Some extension may want to access the Scrapy engine, to inspect or modify the downloader and scheduler behaviour, although this is an advanced use and this API is not yet stable.

#### **spider**

Spider currently being crawled. This is an instance of the spider class provided while constructing the crawler, and it is created after the arguments given in the [crawl\(\)](#) method.

#### **crawl** (\*args, \*\*kwargs)

Starts the crawler by instantiating its spider class with the given *args* and *kwargs* arguments, while setting the execution engine in motion.

Returns a deferred that is fired when the crawl is finished.

## 6.5.2 Settings API

### `scrapy.settings.SETTINGS_PRIORITIES`

Dictionary that sets the key name and priority level of the default settings priorities used in Scrapy.

Each item defines a settings entry point, giving it a code name for identification and an integer priority. Greater priorities take more precedence over lesser ones when setting and retrieving values in the `Settings` class.

```
SETTINGS_PRIORITIES = {
    'default': 0,
    'command': 10,
    'project': 20,
    'spider': 30,
    'cmdline': 40,
}
```

For a detailed explanation on each settings sources, see: [Settings](#).

## 6.5.3 SpiderLoader API

### `class scrapy.loader.SpiderLoader`

This class is in charge of retrieving and handling the spider classes defined across the project.

Custom spider loaders can be employed by specifying their path in the `:setting:'SPIDER_LOADER_CLASS'` project setting. They must fully implement the `scrapy.interfaces.ISpiderLoader` interface to guarantee an errorless execution.

#### `from_settings` (settings)

This class method is used by Scrapy to create an instance of the class. It's called with the current project settings, and it loads the spiders found recursively in the modules of the `:setting:'SPIDER_MODULES'` setting.

**Parameters** `settings` (`Settings` instance) – project settings

**load** (*spider\_name*)

Get the Spider class with the given name. It'll look into the previously loaded spiders for a spider class with name *spider\_name* and will raise a `KeyError` if not found.

**Parameters** *spider\_name* (*str*) – spider class name

**list** ()

Get the names of the available spiders in the project.

**find\_by\_request** (*request*)

List the spiders' names that can handle the given request. Will try to match the request's url against the domains of the spiders.

**Parameters** *request* (*Request* instance) – queried request

## 6.5.4 Signals API

### 6.5.5 Stats Collector API

There are several Stats Collectors available under the `scrapy.statscollectors` module and they all implement the Stats Collector API defined by the `StatsCollector` class (which they all inherit from).

**class** scrapy.statscollectors.**StatsCollector**

**get\_value** (*key*, *default=None*)

Return the value for the given stats key or default if it doesn't exist.

**get\_stats** ()

Get all stats from the currently running spider as a dict.

**set\_value** (*key*, *value*)

Set the given value for the given stats key.

**set\_stats** (*stats*)

Override the current stats with the dict passed in *stats* argument.

**inc\_value** (*key*, *count=1*, *start=0*)

Increment the value of the given stats key, by the given count, assuming the start value given (when it's not set).

**max\_value** (*key*, *value*)

Set the given value for the given key only if current value for the same key is lower than value. If there is no current value for the given key, the value is always set.

**min\_value** (*key*, *value*)

Set the given value for the given key only if current value for the same key is greater than value. If there is no current value for the given key, the value is always set.

**clear\_stats** ()

Clear all stats.

The following methods are not part of the stats collection api but instead used when implementing custom stats collectors:

**open\_spider** (*spider*)

Open the given spider for stats collection.

**close\_spider** (*spider*)

Close the given spider. After this is called, no more specific stats can be accessed or collected.



## 6.6 Signals

Scrapy uses signals extensively to notify when certain events occur. You can catch some of those signals in your Scrapy project (using an [extension](#), for example) to perform additional tasks or extend Scrapy to add functionality not provided out of the box.

Even though signals provide several arguments, the handlers that catch them don't need to accept all of them - the signal dispatching mechanism will only deliver the arguments that the handler receives.

You can connect to signals (or send your own) through the [Signals API](#).

Here is a simple example showing how you can catch signals and perform some action:

```
from scrapy import signals
from scrapy import Spider

class DmozSpider(Spider):
    name = "dmoz"
    allowed_domains = ["dmoz.org"]
    start_urls = [
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Books/",
        "http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/",
    ]

    @classmethod
    def from_crawler(cls, crawler, *args, **kwargs):
        spider = super(DmozSpider, cls).from_crawler(crawler, *args, **kwargs)
        crawler.signals.connect(spider.spider_closed, signal=signals.spider_closed)
        return spider

    def spider_closed(self, spider):
        spider.logger.info('Spider closed: %s', spider.name)

    def parse(self, response):
        pass
```

### 6.6.1 Deferred signal handlers

Some signals support returning [Twisted deferreds](#) from their handlers, see the [Built-in signals reference](#) below to know which ones.

### 6.6.2 Built-in signals reference

Here's the list of Scrapy built-in signals and their meaning.

#### engine\_started

`scrapy.signals.engine_started()`

Sent when the Scrapy engine has started crawling.

This signal supports returning deferreds from their handlers.

---

**Note:** This signal may be fired *after* the `:signal:'spider_opened'` signal, depending on how the spider was started. So **don't** rely on this signal getting fired before `:signal:'spider_opened'`.

---

## engine\_stopped

`scrapy.signals.engine_stopped()`

Sent when the Scrapy engine is stopped (for example, when a crawling process has finished).

This signal supports returning deferreds from their handlers.

## item\_scraped

`scrapy.signals.item_scraped(item, response, spider)`

Sent when an item has been scraped, after it has passed all the *Item Pipeline* stages (without being dropped).

This signal supports returning deferreds from their handlers.

### Parameters

- **item** (dict or *Item* object) – the item scraped
- **spider** (*Spider* object) – the spider which scraped the item
- **response** (*Response* object) – the response from where the item was scraped

## item\_dropped

`scrapy.signals.item_dropped(item, response, exception, spider)`

Sent after an item has been dropped from the *Item Pipeline* when some stage raised a *DropItem* exception.

This signal supports returning deferreds from their handlers.

### Parameters

- **item** (dict or *Item* object) – the item dropped from the *Item Pipeline*
- **spider** (*Spider* object) – the spider which scraped the item
- **response** (*Response* object) – the response from where the item was dropped
- **exception** (*DropItem* exception) – the exception (which must be a *DropItem* subclass) which caused the item to be dropped

## item\_error

`scrapy.signals.item_error(item, response, spider, failure)`

Sent when a *Item Pipeline* generates an error (ie. raises an exception), except *DropItem* exception.

This signal supports returning deferreds from their handlers.

### Parameters

- **item** (dict or *Item* object) – the item dropped from the *Item Pipeline*
- **response** (*Response* object) – the response being processed when the exception was raised
- **spider** (*Spider* object) – the spider which raised the exception

- **failure** (*Failure* object) – the exception raised as a Twisted *Failure* object

## spider\_closed

`scrapy.signals.spider_closed(spider, reason)`

Sent after a spider has been closed. This can be used to release per-spider resources reserved on **:signal:'spider\_opened'**.

This signal supports returning deferreds from their handlers.

### Parameters

- **spider** (*Spider* object) – the spider which has been closed
- **reason** (*str*) – a string which describes the reason why the spider was closed. If it was closed because the spider has completed scraping, the reason is `'finished'`. Otherwise, if the spider was manually closed by calling the `close_spider` engine method, then the reason is the one passed in the `reason` argument of that method (which defaults to `'cancelled'`). If the engine was shutdown (for example, by hitting Ctrl-C to stop it) the reason will be `'shutdown'`.

## spider\_opened

`scrapy.signals.spider_opened(spider)`

Sent after a spider has been opened for crawling. This is typically used to reserve per-spider resources, but can be used for any task that needs to be performed when a spider is opened.

This signal supports returning deferreds from their handlers.

**Parameters** **spider** (*Spider* object) – the spider which has been opened

## spider\_idle

`scrapy.signals.spider_idle(spider)`

Sent when a spider has gone idle, which means the spider has no further:

- requests waiting to be downloaded
- requests scheduled
- items being processed in the item pipeline

If the idle state persists after all handlers of this signal have finished, the engine starts closing the spider. After the spider has finished closing, the **:signal:'spider\_closed'** signal is sent.

You may raise a *DontCloseSpider* exception to prevent the spider from being closed.

This signal does not support returning deferreds from their handlers.

**Parameters** **spider** (*Spider* object) – the spider which has gone idle

---

**Note:** Scheduling some requests in your **:signal:'spider\_idle'** handler does **not** guarantee that it can prevent the spider from being closed, although it sometimes can. That's because the spider may still remain idle if all the scheduled requests are rejected by the scheduler (e.g. filtered due to duplication).

---

## spider\_error

`scrapy.signals.spider_error` (*failure, response, spider*)

Sent when a spider callback generates an error (ie. raises an exception).

This signal does not support returning deferreds from their handlers.

### Parameters

- **failure** (*Failure* object) – the exception raised as a Twisted *Failure* object
- **response** (*Response* object) – the response being processed when the exception was raised
- **spider** (*Spider* object) – the spider which raised the exception

## request\_scheduled

`scrapy.signals.request_scheduled` (*request, spider*)

Sent when the engine schedules a *Request*, to be downloaded later.

The signal does not support returning deferreds from their handlers.

### Parameters

- **request** (*Request* object) – the request that reached the scheduler
- **spider** (*Spider* object) – the spider that yielded the request

## request\_dropped

`scrapy.signals.request_dropped` (*request, spider*)

Sent when a *Request*, scheduled by the engine to be downloaded later, is rejected by the scheduler.

The signal does not support returning deferreds from their handlers.

### Parameters

- **request** (*Request* object) – the request that reached the scheduler
- **spider** (*Spider* object) – the spider that yielded the request

## request\_reached\_downloader

`scrapy.signals.request_reached_downloader` (*request, spider*)

Sent when a *Request* reached downloader.

The signal does not support returning deferreds from their handlers.

### Parameters

- **request** (*Request* object) – the request that reached downloader
- **spider** (*Spider* object) – the spider that yielded the request

## response\_received

`scrapy.signals.response_received(response, request, spider)`  
Sent when the engine receives a new *Response* from the downloader.

This signal does not support returning deferreds from their handlers.

### Parameters

- **response** (*Response* object) – the response received
- **request** (*Request* object) – the request that generated the response
- **spider** (*Spider* object) – the spider for which the response is intended

## response\_downloaded

`scrapy.signals.response_downloaded(response, request, spider)`  
Sent by the downloader right after a *HTTPResponse* is downloaded.

This signal does not support returning deferreds from their handlers.

### Parameters

- **response** (*Response* object) – the response downloaded
- **request** (*Request* object) – the request that generated the response
- **spider** (*Spider* object) – the spider for which the response is intended

## 6.7 Item Exporters

Once you have scraped your items, you often want to persist or export those items, to use the data in some other application. That is, after all, the whole purpose of the scraping process.

For this purpose Scrapy provides a collection of Item Exporters for different output formats, such as XML, CSV or JSON.

### 6.7.1 Using Item Exporters

If you are in a hurry, and just want to use an Item Exporter to output scraped data see the *Feed exports*. Otherwise, if you want to know how Item Exporters work or need more custom functionality (not covered by the default exports), continue reading below.

In order to use an Item Exporter, you must instantiate it with its required args. Each Item Exporter requires different arguments, so check each exporter documentation to be sure, in *Built-in Item Exporters reference*. After you have instantiated your exporter, you have to:

1. call the method `start_exporting()` in order to signal the beginning of the exporting process
2. call the `export_item()` method for each item you want to export
3. and finally call the `finish_exporting()` to signal the end of the exporting process

Here you can see an *Item Pipeline* which uses multiple Item Exporters to group scraped items to different files according to the value of one of their fields:

```
from scrapy.exporters import XmlItemExporter

class PerYearXmlExportPipeline(object):
    """Distribute items across multiple XML files according to their 'year' field"""

    def open_spider(self, spider):
        self.year_to_exporter = {}

    def close_spider(self, spider):
        for exporter in self.year_to_exporter.values():
            exporter.finish_exporting()
            exporter.file.close()

    def _exporter_for_item(self, item):
        year = item['year']
        if year not in self.year_to_exporter:
            f = open('{}.xml'.format(year), 'wb')
            exporter = XmlItemExporter(f)
            exporter.start_exporting()
            self.year_to_exporter[year] = exporter
        return self.year_to_exporter[year]

    def process_item(self, item, spider):
        exporter = self._exporter_for_item(item)
        exporter.export_item(item)
        return item
```

## 6.7.2 Serialization of item fields

By default, the field values are passed unmodified to the underlying serialization library, and the decision of how to serialize them is delegated to each particular serialization library.

However, you can customize how each field value is serialized *before it is passed to the serialization library*.

There are two ways to customize how a field will be serialized, which are described next.

### 1. Declaring a serializer in the field

If you use *Item* you can declare a serializer in the *field metadata*. The serializer must be a callable which receives a value and returns its serialized form.

Example:

```
import scrapy

def serialize_price(value):
    return '$ %s' % str(value)

class Product(scrapy.Item):
    name = scrapy.Field()
    price = scrapy.Field(serializer=serialize_price)
```

### 2. Overriding the `serialize_field()` method

You can also override the `serialize_field()` method to customize how your field value will be exported.

Make sure you call the base class `serialize_field()` method after your custom code.

Example:

```
from scrapy.exporter import XmlItemExporter

class ProductXmlExporter(XmlItemExporter):

    def serialize_field(self, field, name, value):
        if field == 'price':
            return '$ %s' % str(value)
        return super(Product, self).serialize_field(field, name, value)
```

### 6.7.3 Built-in Item Exporters reference

Here is a list of the Item Exporters bundled with Scrapy. Some of them contain output examples, which assume you're exporting these two items:

```
Item(name='Color TV', price='1200')
Item(name='DVD player', price='200')
```

#### BaseItemExporter

```
class scrapy.exporters.BaseItemExporter (fields_to_export=None,          ex-
                                         port_empty_fields=False,      encoding='utf-8',
                                         indent=0)
```

This is the (abstract) base class for all Item Exporters. It provides support for common features used by all (concrete) Item Exporters, such as defining what fields to export, whether to export empty fields, or which encoding to use.

These features can be configured through the constructor arguments which populate their respective instance attributes: `fields_to_export`, `export_empty_fields`, `encoding`, `indent`.

**export\_item**(*item*)

Exports the given item. This method must be implemented in subclasses.

**serialize\_field**(*field*, *name*, *value*)

Return the serialized value for the given field. You can override this method (in your custom Item Exporters) if you want to control how a particular field or value will be serialized/exported.

By default, this method looks for a serializer *declared in the item field* and returns the result of applying that serializer to the value. If no serializer is found, it returns the value unchanged except for unicode values which are encoded to `str` using the encoding declared in the `encoding` attribute.

#### Parameters

- **field** (*Field* object or an empty dict) – the field being serialized. If a raw dict is being exported (not *Item*) *field* value is an empty dict.
- **name** (*str*) – the name of the field being serialized
- **value** – the value being serialized

**start\_exporting**()

Signal the beginning of the exporting process. Some exporters may use this to generate some required header (for example, the *XmlItemExporter*). You must call this method before exporting any items.

**finish\_exporting()**

Signal the end of the exporting process. Some exporters may use this to generate some required footer (for example, the *XmlItemExporter*). You must always call this method after you have no more items to export.

**fields\_to\_export**

A list with the name of the fields that will be exported, or None if you want to export all fields. Defaults to None.

Some exporters (like *CsvItemExporter*) respect the order of the fields defined in this attribute.

Some exporters may require fields\_to\_export list in order to export the data properly when spiders return dicts (not Item instances).

**export\_empty\_fields**

Whether to include empty/unpopulated item fields in the exported data. Defaults to False. Some exporters (like *CsvItemExporter*) ignore this attribute and always export all empty fields.

This option is ignored for dict items.

**encoding**

The encoding that will be used to encode unicode values. This only affects unicode values (which are always serialized to str using this encoding). Other value types are passed unchanged to the specific serialization library.

**indent**

Amount of spaces used to indent the output on each level. Defaults to 0.

- indent=None selects the most compact representation, all items in the same line with no indentation
- indent<=0 each item on its own line, no indentation
- indent>0 each item on its own line, indented with the provided numeric value

## XmlItemExporter

```
class scrapy.exporters.XmlItemExporter(file, item_element='item', root_element='items',  
                                       **kwargs)
```

Exports Items in XML format to the specified file object.

**Parameters**

- **file** – the file-like object to use for exporting the data. Its write method should accept bytes (a disk file opened in binary mode, a io.BytesIO object, etc)
- **root\_element** (*str*) – The name of root element in the exported XML.
- **item\_element** (*str*) – The name of each item element in the exported XML.

The additional keyword arguments of this constructor are passed to the *BaseItemExporter* constructor.

A typical output of this exporter would be:

```
<?xml version="1.0" encoding="utf-8"?>
<items>
  <item>
    <name>Color TV</name>
    <price>1200</price>
  </item>
  <item>
```

(continues on next page)



(continued from previous page)

```
<name>DVD player</name>
<price>200</price>
</item>
</items>
```

Unless overridden in the `serialize_field()` method, multi-valued fields are exported by serializing each value inside a `<value>` element. This is for convenience, as multi-valued fields are very common.

For example, the item:

```
Item(name=['John', 'Doe'], age='23')
```

Would be serialized as:

```
<?xml version="1.0" encoding="utf-8"?>
<items>
  <item>
    <name>
      <value>John</value>
      <value>Doe</value>
    </name>
    <age>23</age>
  </item>
</items>
```

## CsvItemExporter

**class** scrapy.exporters.CsvItemExporter(*file*, *include\_headers\_line=True*, *join\_multivalued=', ', \*\*kwargs*)

Exports Items in CSV format to the given file-like object. If the `fields_to_export` attribute is set, it will be used to define the CSV columns and their order. The `export_empty_fields` attribute has no effect on this exporter.

### Parameters

- **file** – the file-like object to use for exporting the data. Its `write` method should accept bytes (a disk file opened in binary mode, a `io.BytesIO` object, etc)
- **include\_headers\_line** (*str*) – If enabled, makes the exporter output a header line with the field names taken from `BaseItemExporter.fields_to_export` or the first exported item fields.
- **join\_multivalued** – The char (or chars) that will be used for joining multi-valued fields, if found.

The additional keyword arguments of this constructor are passed to the `BaseItemExporter` constructor, and the leftover arguments to the `csv.writer` constructor, so you can use any `csv.writer` constructor argument to customize this exporter.

A typical output of this exporter would be:

```
product,price
Color TV,1200
DVD player,200
```

## PickleItemExporter

**class** scrapy.exporters.PickleItemExporter (file, protocol=0, \*\*kwargs)

Exports Items in pickle format to the given file-like object.

### Parameters

- **file** – the file-like object to use for exporting the data. Its `write` method should accept bytes (a disk file opened in binary mode, a `io.BytesIO` object, etc)
- **protocol** (*int*) – The pickle protocol to use.

For more information, refer to the [pickle module documentation](#).

The additional keyword arguments of this constructor are passed to the [BaseItemExporter](#) constructor.

Pickle isn't a human readable format, so no output examples are provided.

## PprintItemExporter

**class** scrapy.exporters.PprintItemExporter (file, \*\*kwargs)

Exports Items in pretty print format to the specified file object.

**Parameters** **file** – the file-like object to use for exporting the data. Its `write` method should accept bytes (a disk file opened in binary mode, a `io.BytesIO` object, etc)

The additional keyword arguments of this constructor are passed to the [BaseItemExporter](#) constructor.

A typical output of this exporter would be:

```
{'name': 'Color TV', 'price': '1200'}
{'name': 'DVD player', 'price': '200'}
```

Longer lines (when present) are pretty-formatted.

## JsonItemExporter

**class** scrapy.exporters.JsonItemExporter (file, \*\*kwargs)

Exports Items in JSON format to the specified file-like object, writing all objects as a list of objects. The additional constructor arguments are passed to the [BaseItemExporter](#) constructor, and the leftover arguments to the [JSONEncoder](#) constructor, so you can use any [JSONEncoder](#) constructor argument to customize this exporter.

**Parameters** **file** – the file-like object to use for exporting the data. Its `write` method should accept bytes (a disk file opened in binary mode, a `io.BytesIO` object, etc)

A typical output of this exporter would be:

```
[{"name": "Color TV", "price": "1200"},
{"name": "DVD player", "price": "200"}]
```

**Warning:** JSON is very simple and flexible serialization format, but it doesn't scale well for large amounts of data since incremental (aka. stream-mode) parsing is not well supported (if at all) among JSON parsers (on any language), and most of them just parse the entire object in memory. If you want the power and simplicity of JSON with a more stream-friendly format, consider using [JsonLinesItemExporter](#) instead, or splitting the output in multiple chunks.

## JsonLinesItemExporter

**class** scrapy.exporters.JsonLinesItemExporter (file, \*\*kwargs)

Exports Items in JSON format to the specified file-like object, writing one JSON-encoded item per line. The additional constructor arguments are passed to the *BaseItemExporter* constructor, and the leftover arguments to the *JSONEncoder* constructor, so you can use any *JSONEncoder* constructor argument to customize this exporter.

**Parameters** **file** – the file-like object to use for exporting the data. Its `write` method should accept `bytes` (a disk file opened in binary mode, a `io.BytesIO` object, etc)

A typical output of this exporter would be:

```
{ "name": "Color TV", "price": "1200" }
{ "name": "DVD player", "price": "200" }
```

Unlike the one produced by *JsonItemExporter*, the format produced by this exporter is well suited for serializing large amounts of data.

*Architecture overview* Understand the Scrapy architecture.

*Downloader Middleware* Customize how pages get requested and downloaded.

*Spider Middleware* Customize the input and output of your spiders.

*Extensions* Extend Scrapy with your custom functionality

*Core API* Use it on extensions and middlewares to extend Scrapy functionality

*Signals* See all available signals and how to work with them.

*Item Exporters* Quickly export your scraped items to a file (XML, CSV, etc).



### S

`scrapy.contracts`, 135  
`scrapy.contracts.default`, 135  
`scrapy.crawler`, 194  
`scrapy.downloadermiddlewares`, 171  
`scrapy.downloadermiddlewares.ajaxcrawl`, 182  
`scrapy.downloadermiddlewares.cookies`, 172  
`scrapy.downloadermiddlewares.defaultheaders`, 174  
`scrapy.downloadermiddlewares.downloadtimeout`, 174  
`scrapy.downloadermiddlewares.httppauth`, 174  
`scrapy.downloadermiddlewares.httpcache`, 174  
`scrapy.downloadermiddlewares.httpcompression`, 179  
`scrapy.downloadermiddlewares.httpproxy`, 179  
`scrapy.downloadermiddlewares.redirect`, 180  
`scrapy.downloadermiddlewares.retry`, 181  
`scrapy.downloadermiddlewares.robotstxt`, 182  
`scrapy.downloadermiddlewares.stats`, 182  
`scrapy.downloadermiddlewares.useragent`, 182  
`scrapy.exceptions`, 111  
`scrapy.exporters`, 201  
`scrapy.extensions.closespider`, 192  
`scrapy.extensions.corestats`, 191  
`scrapy.extensions.debug`, 193  
`scrapy.extensions.logstats`, 191  
`scrapy.extensions.memdebug`, 192  
`scrapy.extensions.memusage`, 191  
`scrapy.extensions.statsmailer`, 193  
`scrapy.extensions.telnet`, 122  
`scrapy.http`, 81  
`scrapy.item`, 54  
`scrapy.linkextractors`, 92  
`scrapy.linkextractors.lxmlhtml`, 92  
`scrapy.loader`, 58  
`scrapy.loader.processors`, 66  
`scrapy.mail`, 120  
`scrapy.pipelines.files`, 157  
`scrapy.pipelines.images`, 158  
`scrapy.selector`, 51  
`scrapy.settings`, 195  
`scrapy.signals`, 197  
`scrapy.spidermiddlewares`, 184  
`scrapy.spidermiddlewares.depth`, 185  
`scrapy.spidermiddlewares.httperror`, 186  
`scrapy.spidermiddlewares.offsite`, 187  
`scrapy.spidermiddlewares.referer`, 187  
`scrapy.spidermiddlewares.urllength`, 188  
`scrapy.spiders`, 32  
`scrapy.statscollectors`, 119  
`scrapy.utils.log`, 118  
`scrapy.utils.trackref`, 149



## Symbols

`__nonzero__()` (scrapy.selector.Selector method), 52

## A

`adapt_response()` (scrapy.spiders.XMLFeedSpider method), 38

`add_css()` (scrapy.loader.ItemLoader method), 63

`add_value()` (scrapy.loader.ItemLoader method), 62

`add_xpath()` (scrapy.loader.ItemLoader method), 63

`adjust_request_args()` (scrapy.contracts.Contract method), 135

`AjaxCrawlMiddleware` (class in scrapy.downloadermiddlewares.ajaxcrawl), 182

`allowed_domains` (scrapy.spiders.Spider attribute), 32

## B

`BaseItemExporter` (class in scrapy.exporters), 203

`body` (scrapy.http.Request attribute), 83

`body` (scrapy.http.Response attribute), 89

`body_as_unicode()` (scrapy.http.TextResponse method), 91

## C

`clear_stats()` (scrapy.statscollectors.StatsCollector method), 196

`close_spider()`, 73

`close_spider()` (scrapy.statscollectors.StatsCollector method), 196

`closed()` (scrapy.spiders.Spider method), 34

`CloseSpider`, 112

`Compose` (class in scrapy.loader.processors), 67

`context` (scrapy.loader.ItemLoader attribute), 64

`Contract` (class in scrapy.contracts), 135

`CookiesMiddleware` (class in scrapy.downloadermiddlewares.cookies), 172

`copy()` (scrapy.http.Request method), 83

`copy()` (scrapy.http.Response method), 90

`CoreStats` (class in scrapy.extensions.corestats), 191

`crawl()` (scrapy.crawler.Crawler method), 195

`Crawler` (class in scrapy.crawler), 194

`crawler` (scrapy.spiders.Spider attribute), 33

`CrawlSpider` (class in scrapy.spiders), 36

`css()` (scrapy.http.TextResponse method), 91

`css()` (scrapy.selector.Selector method), 52

`css()` (scrapy.selector.SelectorList method), 52

`CSVFeedSpider` (class in scrapy.spiders), 39

`CsvItemExporter` (class in scrapy.exporters), 205

`custom_settings` (scrapy.spiders.Spider attribute), 33

## D

`default_input_processor` (scrapy.loader.ItemLoader attribute), 64

`default_item_class` (scrapy.loader.ItemLoader attribute), 64

`default_output_processor` (scrapy.loader.ItemLoader attribute), 64

`default_selector_class` (scrapy.loader.ItemLoader attribute), 64

`DefaultHeadersMiddleware` (class in scrapy.downloadermiddlewares.defaultheaders), 174

`delimiter` (scrapy.spiders.CSVFeedSpider attribute), 39

`DepthMiddleware` (class in scrapy.spidermiddlewares.depth), 185

`DontCloseSpider`, 112

`DownloaderMiddleware` (class in scrapy.downloadermiddlewares), 171

`DownloaderStats` (class in scrapy.downloadermiddlewares.stats), 182

`DownloadTimeoutMiddleware` (class in scrapy.downloadermiddlewares.downloadtimeout), 174

`DropItem`, 112

`DummyStatsCollector` (class in scrapy.statscollectors), 120

## E

encoding (scrapy.exporters.BaseItemExporter attribute), 204

encoding (scrapy.http.TextResponse attribute), 91

engine (scrapy.crawler.Crawler attribute), 195

engine\_started() (in module scrapy.signals), 197

engine\_stopped() (in module scrapy.signals), 198

export\_empty\_fields (scrapy.exporters.BaseItemExporter attribute), 204

export\_item() (scrapy.exporters.BaseItemExporter method), 203

extensions (scrapy.crawler.Crawler attribute), 195

extract() (scrapy.selector.Selector method), 52

extract() (scrapy.selector.SelectorList method), 53

## F

Field (class in scrapy.item), 58

fields (scrapy.item.Item attribute), 57

fields\_to\_export (scrapy.exporters.BaseItemExporter attribute), 204

FilesPipeline (class in scrapy.pipelines.files), 157

find\_by\_request() (scrapy.loader.SpiderLoader method), 196

finish\_exporting() (scrapy.exporters.BaseItemExporter method), 203

flags (scrapy.http.Response attribute), 90

FormRequest (class in scrapy.http), 87

from\_crawler(), 73

from\_crawler() (scrapy.downloadermiddlewares.DownloaderMiddleware method), 172

from\_crawler() (scrapy.spidermiddlewares.SpiderMiddleware method), 185

from\_crawler() (scrapy.spiders.Spider method), 33

from\_response() (scrapy.http.FormRequest class method), 87

from\_settings() (scrapy.loader.SpiderLoader method), 195

from\_settings() (scrapy.mail.MailSender class method), 121

## G

get\_collected\_values() (scrapy.loader.ItemLoader method), 64

get\_css() (scrapy.loader.ItemLoader method), 63

get\_input\_processor() (scrapy.loader.ItemLoader method), 64

get\_media\_requests() (scrapy.pipelines.files.FilesPipeline method), 157

get\_media\_requests() (scrapy.pipelines.images.ImagesPipeline method), 158

get\_oldest() (in module scrapy.utils.trackref), 150

get\_output\_processor() (scrapy.loader.ItemLoader method), 64

get\_output\_value() (scrapy.loader.ItemLoader method), 64

get\_stats() (scrapy.statscollectors.StatsCollector method), 196

get\_value() (scrapy.loader.ItemLoader method), 62

get\_value() (scrapy.statscollectors.StatsCollector method), 196

get\_xpath() (scrapy.loader.ItemLoader method), 62

## H

headers (scrapy.http.Request attribute), 83

headers (scrapy.http.Response attribute), 89

headers (scrapy.spiders.CSVFeedSpider attribute), 39

HtmlResponse (class in scrapy.http), 91

HttpAuthMiddleware (class in scrapy.downloadermiddlewares.httppauth), 174

HttpCacheMiddleware (class in scrapy.downloadermiddlewares.httpcache), 174

HttpCompressionMiddleware (class in scrapy.downloadermiddlewares.httpcompression), 179

HttpErrorMiddleware (class in scrapy.spidermiddlewares.httperror), 186

HttpProxyMiddleware (class in scrapy.downloadermiddlewares.httpproxy), 179

## I

Identity (class in scrapy.loader.processors), 66

IgnoreRequest, 112

ImagesPipeline (class in scrapy.pipelines.images), 158

inc\_value() (scrapy.statscollectors.StatsCollector method), 196

indent (scrapy.exporters.BaseItemExporter attribute), 204

Item (class in scrapy.item), 57

item (scrapy.loader.ItemLoader attribute), 64

item\_completed() (scrapy.pipelines.files.FilesPipeline method), 157

item\_completed() (scrapy.pipelines.images.ImagesPipeline method), 158

item\_dropped() (in module scrapy.signals), 198

item\_error() (in module scrapy.signals), 198

item\_scraped() (in module scrapy.signals), 198

ItemLoader (class in scrapy.loader), 62

iter\_all() (in module scrapy.utils.trackref), 150

iterator (scrapy.spiders.XMLFeedSpider attribute), 38

itertag (scrapy.spiders.XMLFeedSpider attribute), 38

## J

Join (class in scrapy.loader.processors), 67

JsonItemExporter (class in scrapy.exporters), 206

JsonLinesItemExporter (class in scrapy.exporters), 207



## L

list() (scrapy.loader.SpiderLoader method), 196  
 load() (scrapy.loader.SpiderLoader method), 195  
 load\_item() (scrapy.loader.ItemLoader method), 64  
 log() (scrapy.spiders.Spider method), 34  
 logger (scrapy.spiders.Spider attribute), 33  
 LogStats (class in scrapy.extensions.logstats), 191  
 LxmlLinkExtractor (class in scrapy.linkextractors.lxmlhtml), 92

## M

MailSender (class in scrapy.mail), 120  
 MapCompose (class in scrapy.loader.processors), 67  
 max\_value() (scrapy.statscollectors.StatsCollector method), 196  
 MemoryStatsCollector (class in scrapy.statscollectors), 120  
 meta (scrapy.http.Request attribute), 83  
 meta (scrapy.http.Response attribute), 89  
 MetaRefreshMiddleware (class in scrapy.downloadermiddlewares.redirect), 180  
 method (scrapy.http.Request attribute), 83  
 min\_value() (scrapy.statscollectors.StatsCollector method), 196

## N

name (scrapy.spiders.Spider attribute), 32  
 namespaces (scrapy.spiders.XMLFeedSpider attribute), 38  
 nested\_css() (scrapy.loader.ItemLoader method), 64  
 nested\_xpath() (scrapy.loader.ItemLoader method), 64  
 NotConfigured, 112  
 NotSupported, 113

## O

object\_ref (class in scrapy.utils.trackref), 149  
 OffsiteMiddleware (class in scrapy.spidermiddlewares.offsite), 187  
 open\_spider(), 73  
 open\_spider() (scrapy.statscollectors.StatsCollector method), 196

## P

parse() (scrapy.spiders.Spider method), 34  
 parse\_node() (scrapy.spiders.XMLFeedSpider method), 38  
 parse\_row() (scrapy.spiders.CSVFeedSpider method), 39  
 parse\_start\_url() (scrapy.spiders.CrawlSpider method), 36  
 PickleItemExporter (class in scrapy.exporters), 206  
 post\_process() (scrapy.contracts.Contract method), 135  
 PprintItemExporter (class in scrapy.exporters), 206

pre\_process() (scrapy.contracts.Contract method), 135  
 print\_live\_refs() (in module scrapy.utils.trackref), 149  
 process\_exception() (scrapy.downloadermiddlewares.DownloaderMiddleware method), 172  
 process\_item(), 73  
 process\_request() (scrapy.downloadermiddlewares.DownloaderMiddleware method), 171  
 process\_response() (scrapy.downloadermiddlewares.DownloaderMiddleware method), 171  
 process\_results() (scrapy.spiders.XMLFeedSpider method), 39  
 process\_spider\_exception() (scrapy.spidermiddlewares.SpiderMiddleware method), 184  
 process\_spider\_input() (scrapy.spidermiddlewares.SpiderMiddleware method), 184  
 process\_spider\_output() (scrapy.spidermiddlewares.SpiderMiddleware method), 184  
 process\_start\_requests() (scrapy.spidermiddlewares.SpiderMiddleware method), 185

## Q

quotechar (scrapy.spiders.CSVFeedSpider attribute), 39

## R

re() (scrapy.selector.Selector method), 52  
 re() (scrapy.selector.SelectorList method), 53  
 RedirectMiddleware (class in scrapy.downloadermiddlewares.redirect), 180  
 RefererMiddleware (class in scrapy.spidermiddlewares.referer), 187  
 register\_namespace() (scrapy.selector.Selector method), 52  
 remove\_namespaces() (scrapy.selector.Selector method), 52  
 replace() (scrapy.http.Request method), 83  
 replace() (scrapy.http.Response method), 90  
 replace\_css() (scrapy.loader.ItemLoader method), 64  
 replace\_value() (scrapy.loader.ItemLoader method), 62  
 replace\_xpath() (scrapy.loader.ItemLoader method), 63  
 Request (class in scrapy.http), 82  
 request (scrapy.http.Response attribute), 89  
 request\_dropped() (in module scrapy.signals), 200  
 request\_reached\_downloader() (in module scrapy.signals), 200  
 request\_scheduled() (in module scrapy.signals), 200  
 Response (class in scrapy.http), 89  
 response\_downloaded() (in module scrapy.signals), 201  
 response\_received() (in module scrapy.signals), 201  
 RetryMiddleware (class in scrapy.downloadermiddlewares.retry), 181  
 ReturnsContract (class in scrapy.contracts.default), 135

RobotsTxtMiddleware (class in scrapy.downloadermiddlewares.robotstxt), 182

Rule (class in scrapy.spiders), 37

rules (scrapy.spiders.CrawlSpider attribute), 36

## S

ScrapesContract (class in scrapy.contracts.default), 135

scrapy.contracts (module), 135

scrapy.contracts.default (module), 135

scrapy.crawler (module), 194

scrapy.downloadermiddlewares (module), 171

scrapy.downloadermiddlewares.ajaxcrawl (module), 182

scrapy.downloadermiddlewares.cookies (module), 172

scrapy.downloadermiddlewares.defaultheaders (module), 174

scrapy.downloadermiddlewares.downloadtimeout (module), 174

scrapy.downloadermiddlewares.httpauth (module), 174

scrapy.downloadermiddlewares.httppcache (module), 174

scrapy.downloadermiddlewares.httpcompression (module), 179

scrapy.downloadermiddlewares.httpproxy (module), 179

scrapy.downloadermiddlewares.redirect (module), 180

scrapy.downloadermiddlewares.retry (module), 181

scrapy.downloadermiddlewares.robotstxt (module), 182

scrapy.downloadermiddlewares.stats (module), 182

scrapy.downloadermiddlewares.useragent (module), 182

scrapy.exceptions (module), 111

scrapy.exporters (module), 201

scrapy.extensions.closespider (module), 192

scrapy.extensions.closespider.CloseSpider (class in scrapy.extensions.closespider), 192

scrapy.extensions.corestats (module), 191

scrapy.extensions.debug (module), 193

scrapy.extensions.debug.Debuggger (class in scrapy.extensions.debug), 194

scrapy.extensions.debug.StackTraceDump (class in scrapy.extensions.debug), 193

scrapy.extensions.logstats (module), 191

scrapy.extensions.memdebug (module), 192

scrapy.extensions.memdebug.MemoryDebugger (class in scrapy.extensions.memdebug), 192

scrapy.extensions.memusage (module), 191

scrapy.extensions.memusage.MemoryUsage (class in scrapy.extensions.memusage), 191

scrapy.extensions.statsmailer (module), 193

scrapy.extensions.statsmailer.StatsMailer (class in scrapy.extensions.statsmailer), 193

scrapy.extensions.telnet (module), 122, 191

scrapy.extensions.telnet.TelnetConsole (class in scrapy.extensions.telnet), 191

scrapy.http (module), 81

scrapy.item (module), 54

in scrapy.linkextractors (module), 92

scrapy.linkextractors.lxmlhtml (module), 92

scrapy.loader (module), 58, 195

scrapy.loader.processors (module), 66

scrapy.mail (module), 120

scrapy.pipelines.files (module), 157

scrapy.pipelines.images (module), 158

scrapy.selector (module), 51

scrapy.settings (module), 195

scrapy.signals (module), 197

scrapy.spidermiddlewares (module), 184

scrapy.spidermiddlewares.depth (module), 185

scrapy.spidermiddlewares.httperror (module), 186

scrapy.spidermiddlewares.offsite (module), 187

scrapy.spidermiddlewares.referer (module), 187

scrapy.spidermiddlewares.urllength (module), 188

scrapy.spiders (module), 32

scrapy.statscollectors (module), 119, 196

scrapy.utils.log (module), 118

scrapy.utils.trackref (module), 149

SelectJmes (class in scrapy.loader.processors), 68

Selector (class in scrapy.selector), 51

selector (scrapy.http.TextResponse attribute), 91

selector (scrapy.loader.ItemLoader attribute), 64

SelectorList (class in scrapy.selector), 52

send() (scrapy.mail.MailSender method), 121

serialize\_field() (scrapy.exporters.BaseItemExporter method), 203

set\_stats() (scrapy.statscollectors.StatsCollector method), 196

set\_value() (scrapy.statscollectors.StatsCollector method), 196

settings (scrapy.crawler.Crawler attribute), 194

settings (scrapy.spiders.Spider attribute), 33

SETTINGS\_PRIORITIES (in module scrapy.settings), 195

signals (scrapy.crawler.Crawler attribute), 194

sitemap\_alternate\_links (scrapy.spiders.SitemapSpider attribute), 41

sitemap\_follow (scrapy.spiders.SitemapSpider attribute), 40

sitemap\_rules (scrapy.spiders.SitemapSpider attribute), 40

sitemap\_urls (scrapy.spiders.SitemapSpider attribute), 40

SitemapSpider (class in scrapy.spiders), 40

Spider (class in scrapy.spiders), 32

spider (scrapy.crawler.Crawler attribute), 195

spider\_closed() (in module scrapy.signals), 199

spider\_error() (in module scrapy.signals), 200

spider\_idle() (in module scrapy.signals), 199

spider\_opened() (in module scrapy.signals), 199

spider\_stats (scrapy.statscollectors.MemoryStatsCollector attribute), 120

SpiderLoader (class in scrapy.loader), 195

SpiderMiddleware (class in scrapy.spidermiddlewares),  
[184](#)  
 start\_exporting() (scrapy.exporters.BaseItemExporter  
 method), [203](#)  
 start\_requests() (scrapy.spiders.Spider method), [33](#)  
 start\_urls (scrapy.spiders.Spider attribute), [32](#)  
 stats (scrapy.crawler.Crawler attribute), [194](#)  
 StatsCollector (class in scrapy.statscollectors), [196](#)  
 status (scrapy.http.Response attribute), [89](#)

## T

TakeFirst (class in scrapy.loader.processors), [66](#)  
 text (scrapy.http.TextResponse attribute), [90](#)  
 TextResponse (class in scrapy.http), [90](#)

## U

update\_telnet\_vars() (in module scrapy.extensions.telnet),  
[124](#)  
 url (scrapy.http.Request attribute), [83](#)  
 url (scrapy.http.Response attribute), [89](#)  
 UrlContract (class in scrapy.contracts.default), [135](#)  
 urljoin() (scrapy.http.Response method), [90](#)  
 UrlLengthMiddleware (class in  
     scrapy.spidermiddlewares.urllength), [188](#)  
 UserAgentMiddleware (class in  
     scrapy.downloadermiddlewares.useragent),  
[182](#)

## X

XMLFeedSpider (class in scrapy.spiders), [38](#)  
 XmlItemExporter (class in scrapy.exporters), [204](#)  
 XmlResponse (class in scrapy.http), [91](#)  
 xpath() (scrapy.http.TextResponse method), [91](#)  
 xpath() (scrapy.selector.Selector method), [52](#)  
 xpath() (scrapy.selector.SelectorList method), [52](#)